

RL-TR-95-295, Vol IV (of four)  
Final Technical Report  
April 1996



# **ROMULUS, A COMPUTER SECURITY PROPERTIES MODELING ENVIRONMENT: ROMULUS USER'S MANUAL**

Odyssey Research Associates, Inc.

S. Brackin, S. Foley, L. Gong, B. Hartman, A. Heff, G. Hird,  
D. Long, D. McCullough, I. Meisels, D. Rosenthal,  
I. Sutherland, and A. Weitzman

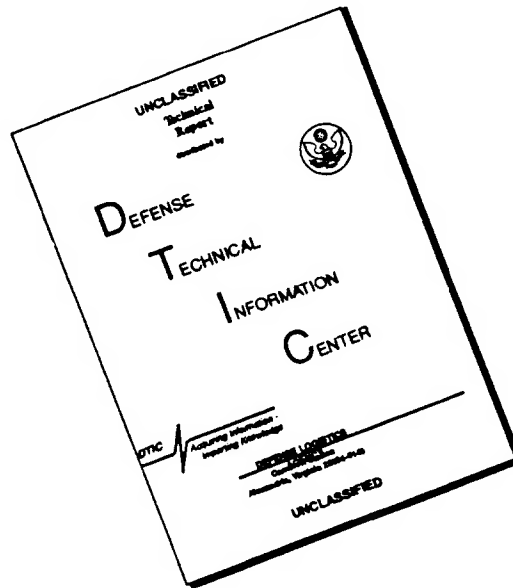
*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

19960724 067

DTIC QUALITY INSPECTED 3

**Rome Laboratory  
Air Force Materiel Command  
Rome, New York**

# DISCLAIMER NOTICE

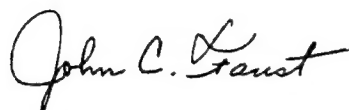


THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be releasable to the general public, including foreign nations.

RL-TR- 95-295, Vol IV (of four), has been reviewed and is approved for publication.

APPROVED:



JOHN C. FAUST  
Project Engineer

FOR THE COMMANDER:



JOHN A. GRANIERO  
Chief Scientist  
Command, Control & Communications Directorate

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify Rome Laboratory ( C3AB), Rome NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE April 1996		3. REPORT TYPE AND DATES COVERED Final Aug 90 - Jun 94
4. TITLE AND SUBTITLE ROMULUS, A COMPUTER SECURITY PROPERTIES MODEL ENVIRONMENT: Romulus User's Manual			5. FUNDING NUMBERS C - F30602-90-C-0092 PE - 35167G PR - 1065 TA - 01 WU - 03	
6. AUTHOR(S) S. Brackin, S. Foley, L. Gong, B. Hartman, A. Heff, G. Hird, D. Long, D. McCullough, I. Meisels, D. Rosenthal, I. Sutherland, and A. Weitzman				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Odyssey Research Associates, Inc. 301 Dates Drive Ithaca NY 14850-1326			8. PERFORMING ORGANIZATION REPORT NUMBER  N/A	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory/C3AB 525 Brooks Rd Rome NY 13441-4505			10. SPONSORING/MONITORING AGENCY REPORT NUMBER  RL-TR-95-295, Vol IV (of four)	
11. SUPPLEMENTARY NOTES  Rome Laboratory Project Engineer: John C. Faust/C3AB/(315) 330-3241				
12a. DISTRIBUTION/AVAILABILITY STATEMENT  Approved for public release; distribution unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words)  The Romulus security properties modeling environment contains tools, theories, and models that support the high-level design and analysis of secure systems.  The Romulus nondisclosure tool supports development and analysis of distributed composite security models and their properties. The Romulus modeling approach establishes the models on a solid theoretical basis and uses formal mathematical tools to aid in the analysis. Romulus allows a user to express a model of a secure system using a formal specification notation that combines graphics and text. Verification of the model proves that it satisfies its critical properties. The user verifies the model by using a combination of automatic decision procedures and interactive theorem proving. The primary emphasis in the current system is the analysis of multilevel trusted system models to see if they satisfy nondisclosure properties. Romulus also includes a tool for formally specifying and verifying authentication protocols. This tool can be used to reason about the beliefs of the parties engaged in a protocol in order to analyze whether the protocol achieves the desired behavior. The Romulus theories include formal theories of nondisclosure, integrity, and (see reverse)				
14. SUBJECT TERMS Computer security, Nondisclosure, Integrity, Availability, Security properties modeling, Information flow analysis, Design verification, Authentication protocol analysis, (see reverse)			15. NUMBER OF PAGES 208 16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT  UL	



---

13. (Cont'd)

availability security. The Romulus library of models demonstrates the application of these theories.

14. (Cont'd)

Multilevel security, Security policy

## Preface

This four volume report describes Romulus, a security modeling environment. Romulus includes a tool for constructing graphical hierarchical process representations; an information flow analyzer; a process specification language; and techniques to aid in doing proofs of security properties. Romulus also contains tools for specifying and analyzing authentication protocols. Using Romulus, a user can develop and analyze security models and properties. The foundations of Romulus are formal theories of security; applications of these theories are demonstrated in a library of models.

The Romulus tool set includes a graphical interface, which is a customizable X11 application that runs under Releases 4 or 5 of X11 or under Release 3 of OpenWindows from Sun Microsystems. It runs on any hardware supporting one of these windowing systems, particularly Sun SPARCstations. The Romulus theories, utilities, and tactics for the HOL prover work under HOL90.

In this volume, we assume familiarity with either X11 or OpenWindows. We also assume general familiarity with developing models and writing formal specifications; we do not assume familiarity with HOL.

## Organization of the Romulus Documentation Set

Volume I of this documentation set is an overview of Romulus. Volume II describes the Romulus theories of nondisclosure, integrity, and availability. Volume III describes the Romulus library of models. This volume is Volume IV; it is the software user's manual. It describes the Romulus tools and gives enough information to try the examples presented in this volume. To begin using Romulus for other models, consult the other sources that are listed in the bibliography of this volume.

---

## Organization of This Volume

This volume has six chapters and two appendices:

- Chapter 1 is an introduction to the Romulus tools.
- Chapter 2 shows how to use the Romulus tools to confirm nondisclosure security for a simple example.
- Chapter 3 gives a full description of the graphical interface and explains each of its commands.
- Chapter 4 introduces the HOL90 environment.
- Chapter 5 describes IPSL, the Romulus interface process specification language, and PSL, the process specification language that IPSL uses; this chapter also provides a tutorial example of the analysis of the nondisclosure properties of a token ring station.
- Chapter 6 describes how to use the Romulus implementation of the logic of authentication to analyze protocols and provides a tutorial example of the Denning-Sacco protocol using the authentication protocol toolkit.
- Appendix A describes the user-customizable parameters of the graphical interface.
- Appendix B describes the Romulus HOL90 library and its contents.

## Conventions

This document set uses the following conventions. Computer code, specifications, program names, file names, and similar material are typeset using a *typewriter* font. Interactive computer sessions are surrounded by a rounded box. Within this box, user input is typeset using an *italic typewriter* font; computer output is typeset using the *typewriter* font. Some computer output has been reformatted for presentation purposes; it may not appear in this document exactly as it appears on your screen.

## **Acknowledgements**

Romulus has been in development for several years. We wish to acknowledge the contributions of everyone who has worked on this project.

Portions of this volume were extracted from previous ORA reports and course materials, specifically, [2], [3], [4], and [7].

# Contents

<b>1</b>	<b>Introduction to the Romulus Tools</b>	<b>1</b>
<b>2</b>	<b>Introduction to the Nondisclosure Tools</b>	<b>5</b>
2.1	The Graphical Interface . . . . .	6
2.1.1	Designing a Model . . . . .	7
2.1.2	Flow Analysis . . . . .	13
2.2	A Trusted Process . . . . .	17
2.2.1	Specifying the Filter . . . . .	17
2.2.2	Proving the Filter Secure . . . . .	20
2.3	A Manifestly Secure Process . . . . .	23
2.3.1	Specifying the Unclassified Process . . . . .	23
2.3.2	Proving the Unclassified Process . . . . .	25
2.4	A Composite Process . . . . .	27
2.4.1	Specifying the Simple Example . . . . .	28
2.4.2	Proving the Simple Example . . . . .	29
2.5	Confirming Proof Completion . . . . .	32
<b>3</b>	<b>The Romulus Graphical Interface</b>	<b>34</b>
3.1	Basic Concepts and Terminology . . . . .	36
3.2	General Interface Principles . . . . .	37
3.2.1	Command Buttons . . . . .	39
3.2.2	Text-Entry Windows . . . . .	39
3.2.3	Message Window . . . . .	40
3.2.4	Canvas Window . . . . .	41
3.2.5	Inert Background Areas . . . . .	42
3.3	Command Levels . . . . .	43
3.4	Top-Level Commands . . . . .	44

3.4.1	modify . . . . .	44
3.4.2	flow . . . . .	45
3.4.3	close . . . . .	46
3.4.4	save . . . . .	46
3.4.5	print . . . . .	47
3.4.6	names . . . . .	47
3.4.7	refresh . . . . .	48
3.4.8	quit . . . . .	48
3.5	Component Commands . . . . .	48
3.5.1	create . . . . .	48
3.5.2	load . . . . .	48
3.5.3	move . . . . .	49
3.5.4	delete . . . . .	50
3.5.5	open . . . . .	50
3.5.6	assume . . . . .	50
3.5.7	ips1 . . . . .	51
3.5.8	spec . . . . .	51
3.5.9	check . . . . .	53
3.5.10	display . . . . .	54
3.6	Port Commands . . . . .	55
3.6.1	create/connect . . . . .	55
3.6.2	move . . . . .	57
3.6.3	delete . . . . .	57
3.6.4	display . . . . .	57
3.6.5	modify . . . . .	58
<b>4</b>	<b>Introduction to the HOL90 Environment</b>	<b>60</b>
4.1	Introduction to Standard ML . . . . .	60
4.2	The HOL Logic . . . . .	66
4.2.1	HOL Terms . . . . .	66
4.2.2	HOL Theories . . . . .	69
4.2.3	Defining HOL Types and Constants . . . . .	73
4.3	Goal Oriented Proof: Tactics and Tacticals . . . . .	77
4.3.1	Goals . . . . .	77
4.3.2	Tactics . . . . .	78
4.3.3	Goal Stack . . . . .	80
4.3.4	Tacticals . . . . .	81

4.3.5	Examples . . . . .	81
4.3.6	More HOL Help . . . . .	92
<b>5</b>	<b>Romulus Security Proofs</b>	<b>93</b>
5.1	Romulus Process Specifications . . . . .	93
5.1.1	The Romulus Interface Process Specification Language . . . . .	94
5.1.2	The Romulus Process Specification Language . . . . .	98
5.1.3	The HOL Embedding . . . . .	100
5.2	Proving Processes Secure . . . . .	107
5.2.1	Trusted Processes . . . . .	107
5.2.2	Manifestly Secure Processes . . . . .	112
5.2.3	Composite Processes . . . . .	112
5.3	Tutorial Example . . . . .	113
5.3.1	Secure Token Ring Station . . . . .	114
5.3.2	Graphical Interface . . . . .	115
5.3.3	Flow Analysis . . . . .	118
5.3.4	Sorter Specification and Proof . . . . .	119
5.3.5	IPSL Specification . . . . .	120
5.3.6	Showing Security Using HOL . . . . .	122
5.3.7	Labeler Specification and Proof . . . . .	129
5.3.8	Confirming Security in the Graphical Interface . . . . .	130
<b>6</b>	<b>The Authentication Protocol Toolkit</b>	<b>133</b>
6.1	Describing and Specifying a Protocol . . . . .	134
6.2	Deriving a Proof . . . . .	136
6.3	Tutorial Example . . . . .	137
<b>A</b>	<b>Graphics Interface Parameters</b>	<b>141</b>
A.1	X Defaults Files . . . . .	142
A.2	Environment Variables . . . . .	143
A.3	Command-Line Options . . . . .	143
A.4	Romulus-Specific Application Resources . . . . .	144
A.4.1	abbreviations . . . . .	144
A.4.2	abortsavefile . . . . .	145
A.4.3	arrowheadrise . . . . .	145
A.4.4	arrowheadrun . . . . .	145

A.4.5	buttonfont	146
A.4.6	comerroredge	146
A.4.7	comnormedge	146
A.4.8	componentfont	146
A.4.9	editfont	147
A.4.10	initial	147
A.4.11	labelfont	147
A.4.12	levelfile	147
A.4.13	logofont	148
A.4.14	mineditwidth	148
A.4.15	porterrorline	148
A.4.16	portfont	149
A.4.17	portnamesdisplayed	149
A.4.18	portnormline	149
A.4.19	searchpath	150
A.4.20	textbuttonfont	150
A.4.21	texteditfont	150
A.4.22	textlabelfont	150
A.4.23	translationstable	151

<b>B</b>	<b>The Romulus HOL90 Library</b>	<b>152</b>
B.1	HOL90 Libraries	152
B.2	On-line Help Files	154
B.3	Utilities and Tactics	155
B.3.1	romcontype	155
B.3.2	romrecord	156
B.3.3	romgetconstant	156
B.3.4	romrtheory	156
B.3.5	BNPSP_rightform_TAC	157
B.3.6	BNPSP_nowritesdown_TAC	157
B.3.7	BNPSP_restrictive_TAC	157
B.3.8	BPSP_rightform_TAC	158
B.3.9	BPSP_invpreserved_TAC	158
B.3.10	BPSP_nowritesdown_TAC	158
B.3.11	BPSP_nolowchange_TAC	158
B.3.12	BPSP_samepath_TAC	159
B.3.13	BPSP_lowresponsesame_TAC	159



B.3.14	BPSP_restrictive_TAC . . . . .	159
B.3.15	ManifestlySecure_TAC . . . . .	160
B.3.16	HookupValid_TAC . . . . .	160
B.4	Theories . . . . .	160
B.4.1	romlemmas . . . . .	161
B.4.2	romproc . . . . .	161
B.4.3	romsecure . . . . .	162
B.4.4	sharedstate . . . . .	167
B.4.5	crypto_90 . . . . .	167
B.5	Code Executed After Loading . . . . .	167
<b>C</b>	<b>User Defined Levels</b>	<b>168</b>
	<b>Bibliography</b>	<b>179</b>
	<b>Index</b>	<b>181</b>

---

# List of Tables

4.1 The HOL Logic . . . . .	66
-----------------------------	----

## List of Figures

2.1	Creating components . . . . .	8
2.2	Naming components . . . . .	9
2.3	Creating ports . . . . .	11
2.4	Setting security limits . . . . .	12
2.5	The simple example . . . . .	14
2.6	Flow analysis of the simple example . . . . .	15
2.7	Flow analysis after assumption . . . . .	16
3.1	Romulus window displaying a token ring station . . . . .	38
5.1	Token ring station . . . . .	116

# Chapter 1

## Introduction to the Romulus Tools

The Romulus environment is a collection of tools for modeling, analyzing, and verifying secure systems. The tools address two broad areas of concern: nondisclosure security and the analysis of authentication (cryptographic) protocols.

For analyzing nondisclosure security, the Romulus tools have two levels:

- A graphical analysis of processes described as interconnected collections of simpler subprocesses. (We call such processes *composite* processes.) The Romulus graphical interface makes it easy to describe such processes, their subprocesses, and the connections between these subprocesses. In addition, the graphical interface makes it easy to attach assumptions or assumptions about the security levels of messages entering or leaving composite processes and their subprocesses, and by flow analysis, to confirm that the full processes are nondisclosure secure if particular subprocesses are nondisclosure secure, the assignments of levels to events are consistent across connections between processes and the assumptions and conclusions about event levels are correct.
- A formal analysis of processes that are not described in terms of simpler subprocesses. (We call such processes *atomic* processes.) Romulus tools, including the graphical interface, a language called the Interface Process Specification Language (IPSL) for specifying processes, a

compiler for translating IPSL into formal specifications in Higher Order Logic (HOL), and tactics for guiding the HOL90 prover in proving nondisclosure security, greatly aid in proving that atomic processes are nondisclosure secure.

The graphical interface's use of flow analysis is implicitly based on the *composability* of *restrictiveness*, the version of nondisclosure security that Romulus uses; the composition of properly connected restrictive processes is itself restrictive.

The Romulus tools for proving atomic processes restrictive assume that these processes are *server* processes, meaning that they wait for input, process each input (possibly producing output), and return to wait for the next input. These tools also assume that inputs to these processes are *buffered*, meaning that they are saved on a buffer until the process is ready to accept them. (This last assumption is not unreasonable in the typical case that processing of one input can be relied upon to finish before the next input.)

More specifically, the current Romulus tools for analyzing nondisclosure security are the following:

- A graphical interface for constructing hierarchical process representations. The interface uses icons to show processes and the ports where information enters or leaves them, and uses arrows to show information flows between ports. The interface's description of a process includes the ranges of message security levels that the process's environment guarantees for input messages and that it requires for output messages.
- An information flow analyzer, invoked through the graphical interface, that identifies potential nondisclosure security failures. This tool shows which atomic subprocesses must be restrictive to guarantee that a full process is restrictive and can help indicate where trusted code must be placed in a system design.
- A simple language, the Interface Process Specification Language (IPSL), for describing processes and a translator for translating IPSL into the more complicated corresponding HOL90 process specifications. This translator also produces goal files, which set the goals that must be proved in HOL90 to show that these processes are restrictive.

- A formal HOL90 theory of atomic processes, a formal language defined in this theory for specifying processes, a formal theory of process security, and utilities for defining types useful in specifying processes. All of these theories and utilities are implemented for the HOL90 proof construction system.
- Tactics that aid in proving goals that show that buffered server processes are restrictive, that manifest security conditions hold, that the interface conditions for processes hold, and that connections between processes are properly made. Some of these tactics use other tactics for proving subgoals. These other tactics can be used independently, avoiding the reproofing of major subgoals in proofs of security for complicated processes.
- A HOL90 utility for communicating results proved in HOL90 back to the graphical interface, allowing it to identify processes as proved secure.

The current Romulus tools for analyzing authentication protocols include the following:

- A protocol specification language and a logic of authentication that can be used to formally specify and analyze authentication protocols.
- A tool implementing the above language and logic that supports proving that authentication protocols meet their formal specifications.

This User's Manual describes the tools that are currently supported in the Romulus security modeling environment.

Chapters 2, 3, and 5 describe the nondisclosure tools. Chapter 2 introduces a simple example and gives the mechanics of using the graphical interface and the other Romulus nondisclosure tools to prove that the example is nondisclosure secure. Chapter 3 gives detailed descriptions of the Romulus graphical interface and the flow analysis tool. Chapter 5 describes IPSL, the Romulus HOL90 process specification language (PSL) that it uses, and the Romulus tactics for proving processes restrictive; this chapter also gives a more complicated example of using the Romulus nondisclosure tools to analyze a system's security. Chapter 4 introduces the HOL90 theorem proving environment and provides background for the material in Chapter 5.

---

Chapter 6 describes how to use the Romulus implementation of the logic of authentication to analyze protocols and provides a tutorial example of verifying the Denning-Sacco protocol using the authentication protocol toolkit. The material on HOL in Chapter 4 is also relevant to these tools.

Appendix A gives details about the user-customizable parameters of the graphical interface. Appendix B describes the Romulus HOL90 library and its contents.

## Chapter 2

# Introduction to the Nondisclosure Tools

This chapter shows how to use Romulus to model, analyze, and verify that a simple system is nondisclosure secure. The example system consists of two subprocesses. One subprocess, a filter, accepts messages of all security levels and filters out those that are not unclassified. The other subprocess is an unclassified process.

This chapter first illustrates how to use the graphical interface to draw a graphical representation of the whole system. It then shows how to use flow analysis to confirm that only the filter subprocess needs to be proved restrictive. It next shows how to create an Interface Process Specification Language (IPSL) specification of the filter subprocess, translate the specification to produce a HOL90 specification of this subprocess, prove that this subprocess is restrictive, and call `romrtheory` to communicate the result back to the graphical interface. Next, it shows a similar process for the other components in the system to prove the conditions necessary to ensure that all the components are properly connected. Finally, the chapter shows how to confirm with the graphical interface that the filter subprocess and the full system are nondisclosure secure. You can follow along at a terminal to construct the example as you read this chapter.

This chapter primarily describes the mechanics of using the Romulus tools, though it does give brief informal descriptions of what the various actions accomplish. Chapters 3 through 5 give more detail on the Romulus tools and on HOL90. Chapter 5 also gives a thorough description of a more



complicated nondisclosure security example.

## 2.1 The Graphical Interface

The graphical interface allows you to graphically create a system model and to analyze the information flows in this model. First we will bring up the Romulus graphical interface, then we will describe creating a model of our simple system.

To start the Romulus graphical interface, enter

```
romulus
```

in a command window, for example, an `xterm` window. (We assume that you are running X11 Release 5 or OpenWindows Release 3, that the Romulus executables are installed, and that your path includes the directory that contains these executables.) Because we did not specify any command-line options, the graphical interface will print the message

```
unable to open theory file levels.rth:
level information not found; default values used.
```

Since we will use the standard Romulus security levels, we can ignore this message. Figure 2.1 shows an example of the Romulus graphical interface.

The Romulus window is divided into three areas: command buttons, a message window, and a canvas area. The command buttons are used to choose different commands, the message area displays information about the command buttons or the current state of command execution, and the canvas area is used to draw models of systems.

To select any of the commands, click the left mouse button on the command button. When you select a command button, the command is activated. Some commands are carried out immediately (e.g., the quit command), but others are carried out each time you enter necessary text and/or perform appropriate mouse actions for as long as the command is active. When a command is active, the message window summarizes the options available with mouse actions. Selecting another command, reselecting the active command, or clicking the right mouse button while the cursor is on the canvas deselects the active command.

### 2.1.1 Designing a Model

The basic objects in a Romulus design are components and ports. Components form a tree structure; the main component being studied corresponds to a system or top-level process, and subcomponents represent processes within that system or top-level process. Ports represent data connections through which data enters or leaves the process represented by a component or subcomponent.

Figure 2.1 shows a component, **(T)**, that comes up as the default component in the Romulus window. **T** is an example of a *tree address*; tree addresses identify unnamed components and are in parentheses, which distinguishes them from component names. The tree address **T** is used for the top-level component.

To create a model of the simple example, we first create the example's subcomponents, one for the message filter and one for the unclassified subprocess, as shown in Figure 2.1. Select (with the left mouse button) the **create** command in the second row of command buttons, the row labeled **Component operations**. Draw the box for each subcomponent by positioning the mouse cursor on the canvas where you want the upper-left corner of the box to be, press and hold the left mouse button, drag the mouse to draw a rectangle, and release the button where you want the lower-right corner of the box to be.

The **(Tc1)** and **(Tc2)** in the subcomponents indicate that neither component is named. The tree address **Tc1** indicates that this component is the first child component (i.e., first-created immediate subcomponent) of the top-level component and the tree address **Tc2** indicates that the other is the second child component.

We next name the top-level component and its subcomponents. Naming the components is necessary in order to have the graphical interface recognize results proved with the HOL90 prover. Select the **modify** command in the top line of buttons. The message window now explains the effects of using the three mouse buttons with this command. The left mouse button selects a component to modify and confirms the modification made to the previously modified component, if there was one. The middle mouse button selects a component and cancels the modification made to the previously modified component. The right mouse button confirms the change to the previously modified component and terminates the **modify** command. Select the **(T)**

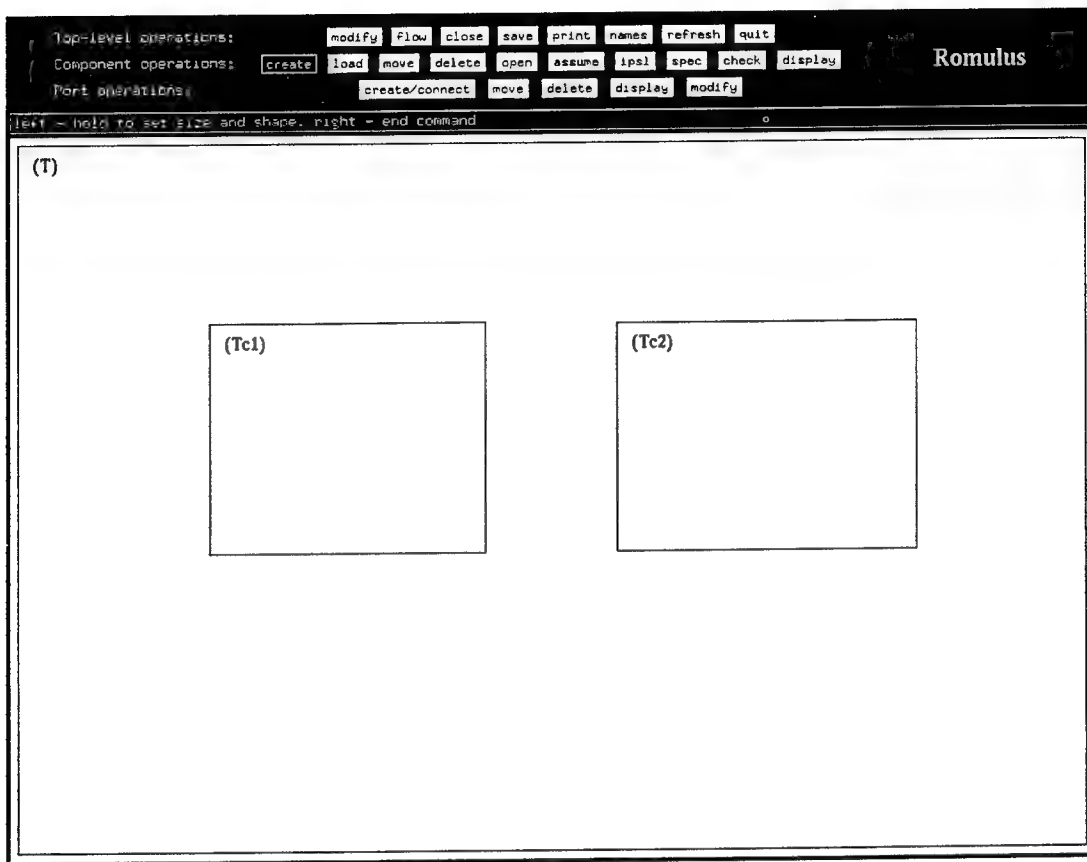


Figure 2.1: Creating components

component by clicking the left mouse button anywhere inside the (T) box but outside the subcomponents' boxes.

Figure 2.2 shows the top-level component selected for modification. When you select a component, a text-entry window appears where you make the selection, and keyboard output is directed to this text-entry window as long as the mouse cursor is in the Romulus window. A caret (^) shows where the next character will be entered. By default, the caret is initially at the beginning of the component name, or as in this case, at the beginning of the word None (which indicates that the component is unnamed).

Use the arrow keys and the backspace key, or the Emacs-style control-d or control-k commands, to erase None; next type the new name, simple\_example. Be careful not to enter a carriage return after the name

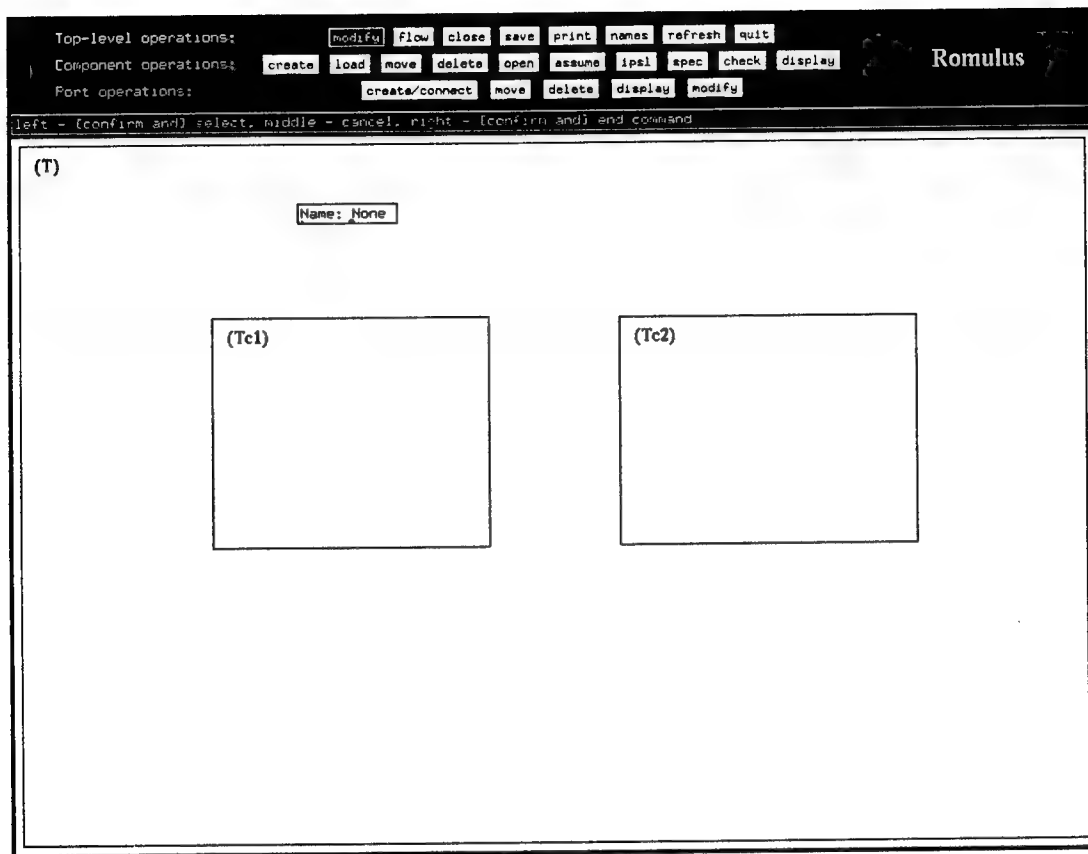


Figure 2.2: Naming components

(Romulus treats it as an error), or to change the label **Name:** or the space separating this label from the name. You can also use the commands **control-b** and **control-f** to edit the text while in the text-entry window.

Now we select a child component by clicking the left mouse button inside its box. Selecting the first child component with the left mouse button confirms the top-level component's **simple.example** name and allows you to enter the name **filter** for this child. Selecting the second child component with the left mouse button confirms the first child's **filter** name and allows you to enter the name **unclassified\_process** for the second child. Clicking the right mouse button confirms the second child's name and ends the **modify** command. Make sure that the cursor is outside the text-entry window when you click the right mouse button to end the command.

We next create the channels through which data flows into and out of our example process and its subprocesses. A *port* icon, on the edge of a component's box, represents an interface through which data enters or leaves the process represented by that component. The connections between ports, drawn as arrows, show how data flows between processes.

We create ports for our example by selecting the **create/connect** command in the **Port operations** line of buttons. Using this command, you can create individual ports and then connect them, or you can create pairs of ports and the connection between them at the same time, which is usually more convenient. Create a connection from the left edge of **simple\_example** into the left edge of **filter** by pushing down the left mouse button inside and near the left edge of **simple\_example**, holding it down and dragging the cursor inside and near the left edge of **filter**, and releasing the button. The command automatically deduces the types of the ports that could be connected to give this data flow, in this case both input ports; input ports are drawn as disks with holes in their centers. You should now have ports as shown in Figure 2.3.

Repeat this process to create connections from the right edge of **filter** to the left edge of **unclassified\_process** and from the right edge of **unclassified\_process** to the right edge of **simple\_example**. The connection from **filter** to **unclassified\_process** will be from an output port to an input port, and the connection between **unclassified\_process** and **simple\_example** will be between two output ports; output ports are drawn as solid diamonds.

The `[_,_]` annotations displayed near each port icon indicate that no upper or lower limits have been set on the security levels of messages passing through that port. We next assign such limits to all ports using the **modify** command in the **Port operations** line of buttons. (The top-level **modify** command can be used to assign security-level limits to ports, but the port-operation **modify** command is usually more convenient for assigning limits.)

Selecting the port **modify** command causes a text-entry window to replace the **Romulus** logo in the upper-right corner of the graphical interface window. You enter the name of a security level in the empty rectangle in this window just as you enter names in the top-level **modify** command's text-entry windows. (The default limit strings are **unclassified**, **confidential**, **secret**, and **top\_secret**.) After entering the level, you indicate whether it is to be used as a low limit or a high limit by selecting either the **low** or **high** button in the text-entry window. For this example, enter

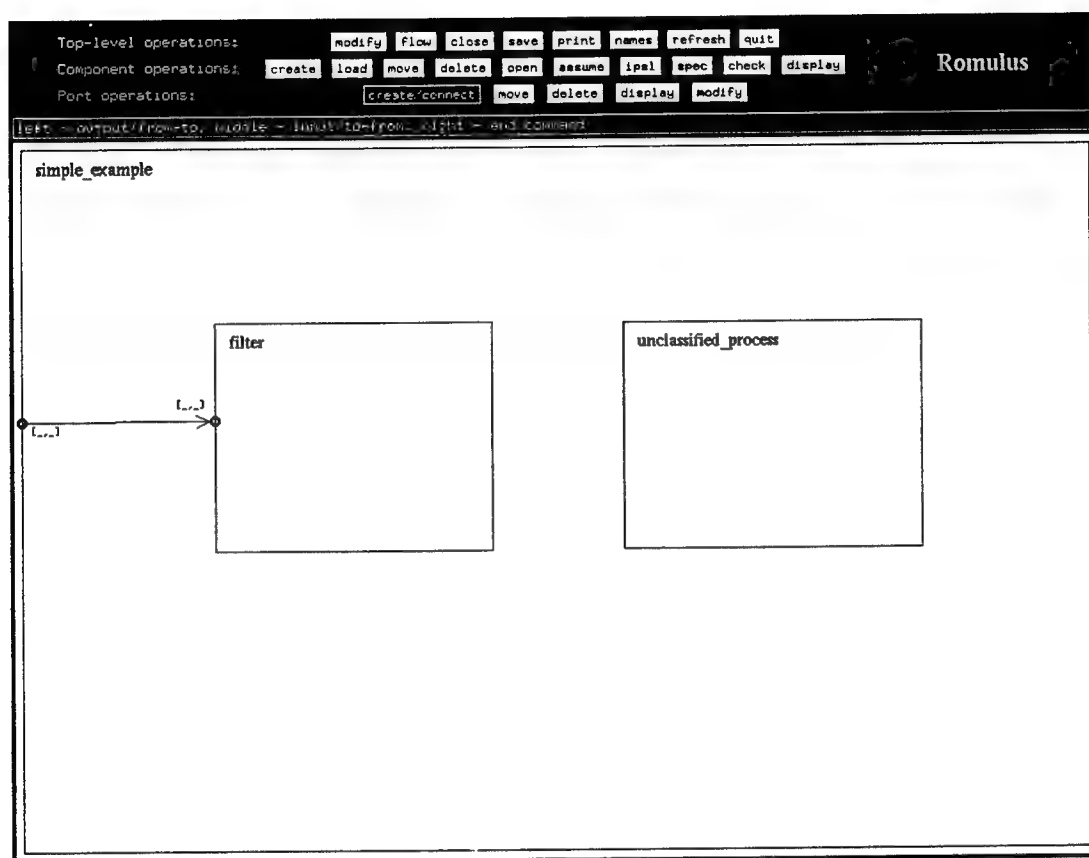


Figure 2.3: Creating ports

unclassified and select low.

Assign this limit to individual ports by clicking on them with the left mouse button or to the ports at both the ends of connections by clicking on these connections with the middle mouse button. Using the middle mouse button is more convenient for making initial limit assignments, but only the left mouse button is capable of overriding earlier assignments. Using the middle mouse button to assign unclassified as the lower limit to the ports on the connection from **simple\_example** to **filter** produces the result shown in Figure 2.4. The U annotations are abbreviations for “unclassified.”

Note that the graphical interface checks that the ranges assigned to each end of a connection are consistent, that is, it checks that the range assigned to the port at the tail of arrow is a subrange of the range assigned to the

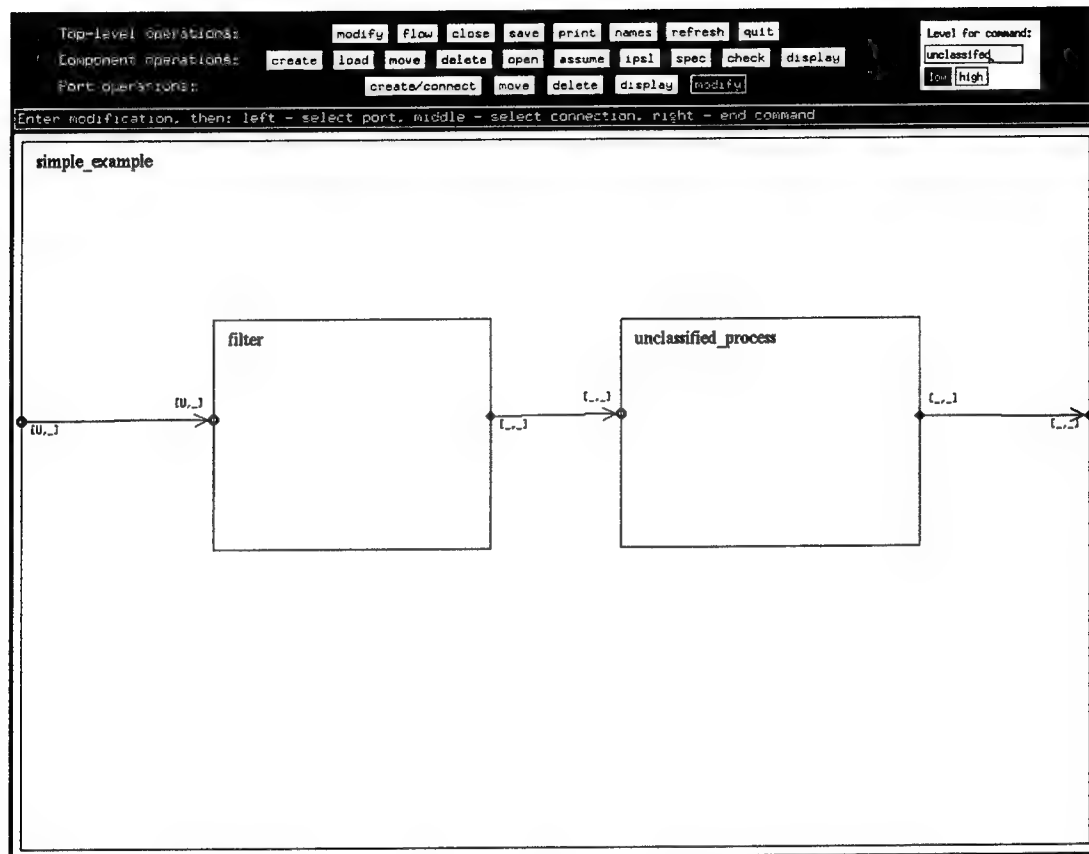


Figure 2.4: Setting security limits

port at the head of the arrow. This may effect the order that Romulus allows you to change the limits on either end of a connection. Also, in a multi-layer model you may find it easier to assign level ranges if you start at the lowest level components and work your way up to higher level components.

We use the same techniques to set the upper security limits for both ports on the connection from **simple\_example** to **filter** to **top\_secret**. Similarly, set the upper and lower security limits for all other ports to **unclassified**.

Finally, we use the **modify** command in the **Top-level operations** line to assign names to all the ports. This command is used in exactly the same way as it was used to give names to components except ports are selected instead of components. (As with components, naming the ports is necessary in order to have the graphical interface recognize results proved in the HOL90

prover.) Since every component in our example has exactly one input port and exactly one output port, name **simple\_example**'s input port **s\_in** and its output port **s\_out**, name **filter**'s input port **f\_in** and its output port **f\_out**, and name **unclassified\_process**'s input port **u\_in** and its output port **u\_out**.

We next display port names next to each port in the canvas window using the **names** command in the Top-level operations row.

Now we have a complete model. You can save it by selecting the top-level **save** command, entering **simple\_example** in its text-entry window, and selecting the **confirm** button in this text-entry window; Figure 2.5 shows the complete example before the **confirm** button is pressed. The **save** command uses four files to save the contents of this example. The file **simple\_example.rom** contains information that describes the sizes and positions of the components and ports. In addition to the **.rom** file, the **save** command produces an **.ips1** file for each component that contains information about names of components and ports, level ranges, connections ports, etc. In this example, the three **.ips1** files created by **save** are **simple\_example.ips1**, **filter.ips1**, and **unclassified\_process.ips1**.

If you were to exit from Romulus at this point (using the quit button), you could restart Romulus with the simple example with the command

```
romulus -initial=simple_example
```

Port names will not initially be displayed; use the **names** command to display the port names.

Do not forget to save your model using the **save** command *before* you exit from Romulus. Also, you should *always* invoke the **save** command from the top-level component. If you invoke **save** from a lower-level component and then exit Romulus, only part of your model will be saved.

## 2.1.2 Flow Analysis

We now begin to establish that the model is secure. The Romulus flow analyzer checks for components that potentially allow insecure data flows. A possible insecure data flow is a path from a port, through at least one component, to another port where the upper security level on the first port is higher than the lower security level on the second port; high-level information might enter the first port and be conveyed in some form to a person or process



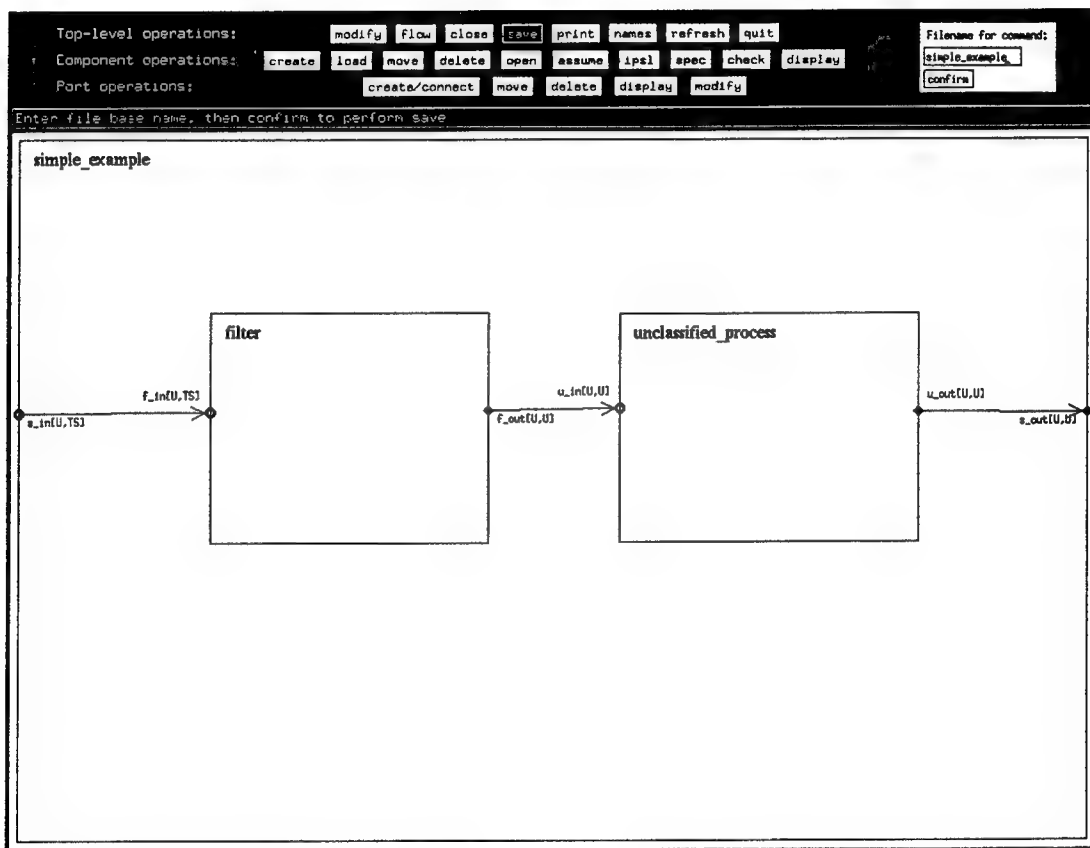


Figure 2.5: The simple example

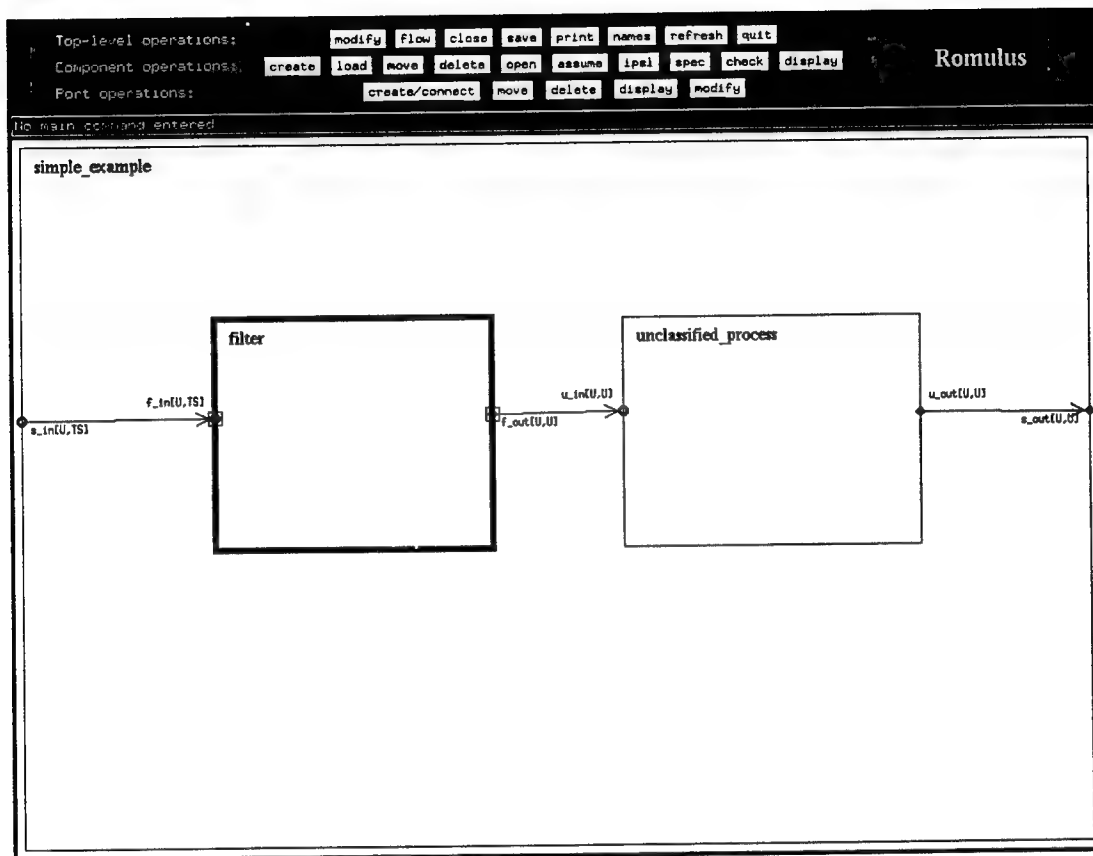


Figure 2.6: Flow analysis of the simple example

not authorized to receive it at the second port. The flow analyzer checks the model for possible insecure data flows and, if it finds one, highlights it by drawing boxes around the ports at its ends and drawing components contained in the flow with bold lines.

Figure 2.6 shows the effect of selecting the **flow** command in the Top-level operations row to do a flow analysis on our simple example. It shows, as one would expect, that top-secret information could potentially enter the **filter** component and emerge to be observed by a user or process with no clearance at all.

Figure 2.7 shows the effect of selecting the **assume** command in the Component operations row and clicking on the **filter** component with the left mouse button to assume that this component is secure. A single asterisk ap-

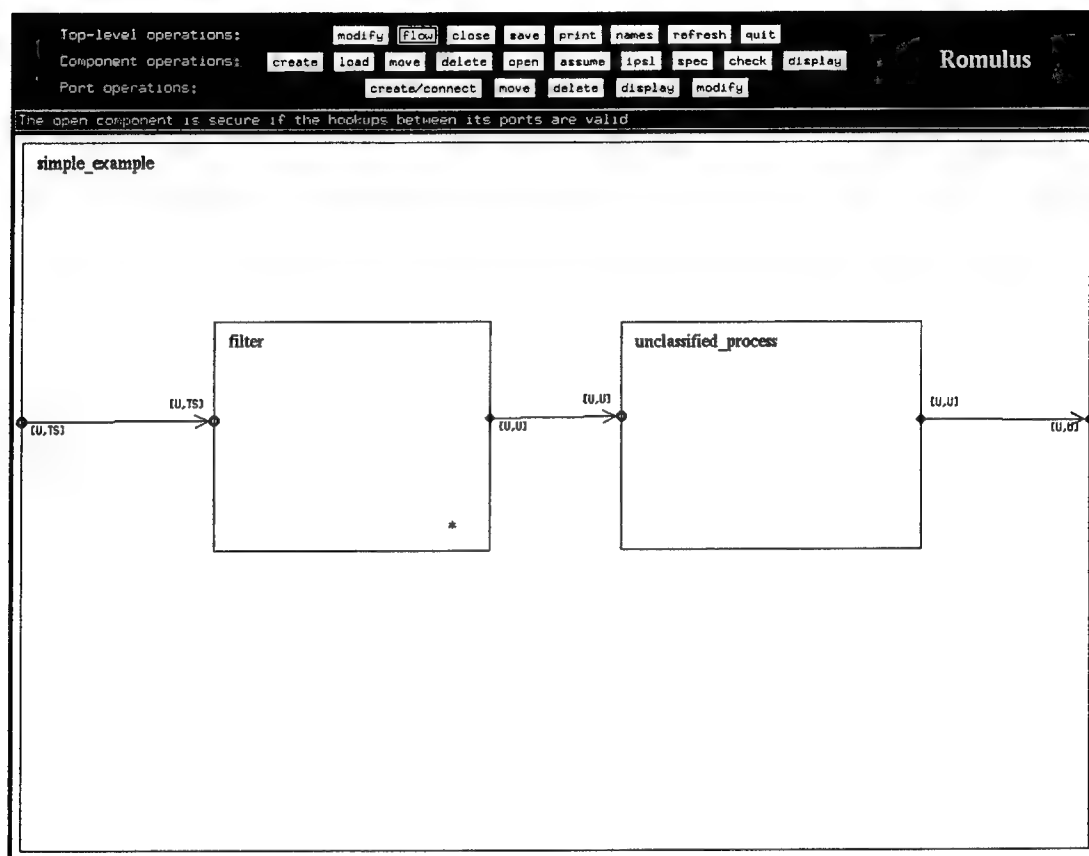


Figure 2.7: Flow analysis after assumption

appears in the bottom-right corner of **filter** to show that it is assumed secure. Now, selecting the **flow** command shows no new insecure flows. Instead a message appears in the message window that indicates (under the assumption that the filter is secure) that simple example is secure if the connections between the ports are valid. The next step is to specify the atomic **filter** component more completely and then prove that it is restrictive. This is the topic of the next section. The validity of connections between ports is considered in sections 2.3 and 2.4.

## 2.2 A Trusted Process

The **filter** process is an example of an atomic process that must be trusted to correctly handle multi-level information. Romulus proves that such processes are secure by proving that they are restrictive under the assumption that the level ranges for their input ports are valid. In addition, it is necessary to show that the assumptions made by the flow analyzer are correct, that is for each process, if all its inputs are in the specified input ranges, then all its outputs will be in the specified output ranges. This section describes how these conditions are proved using the **filter** process as an example.

### 2.2.1 Specifying the Filter

We prepare a formal specification of the **filter** component by first completing an IPSL specification for it, then translating this specification into a HOL90 specification.

We start with the file `filter.ipsl` produced by the `save` command.

```
??Process: filter

??OutPort: f_out
??MessageVar:
??LevelFun:
??LevelRange: unclassified unclassified

??InPort: f_in
??MessageVar:
??LevelFun:
??LevelRange: unclassified top_secret
??Response:

??EndProcess: filter
```

This file is a partial IPSL specification of the **filter** process that contains everything that was specified using the graphical interface.

IPSL is discussed more fully in Chapter 5, but for the moment we note that `??Process:`, `??OutPort:`, and `??InPort:` identify the starts of process, output port, and input port specifications respectively. Messages passing through a port are defined as tuples of variables. A port specification

must have one `??MessageVar:` entry for each variable in the tuple of variables for that port. The `??LevelFun:` value gives the security level, as a function of the variables in the message, of messages passing through the port. The `??LevelRange:` entry gives the range of security levels of messages passing through the port. The `??Response:` value for an input port gives the response, as a function of the variables in an arbitrary message, of the process to messages entering through that port; this response is given using the Romulus formal process specification language, PSL.

We edit `filter.ipsl`, without changing the fields that have already been filled in by the graphical interface, to produce the following file:

```

??Process: filter
??HOL_functions:
  new_parent "string";
  new_constant
    {Name="source_level",
     Ty= ==':string->standard_level'==};

??OutPort: f_out
??MessageVar: source:string
??MessageVar: data:string
??LevelFun: unclassified
??LevelRange: unclassified unclassified

??InPort: f_in
??MessageVar: source:string
??MessageVar: data:string
??LevelFun: source_level source
??LevelRange: unclassified top_secret
??Response:
(If ((source_level source) = unclassified)
  (Send (f_out source data))
  Skip);;
(Call filterTop)

??EndProcess: filter

```

The main additions here are the `??HOL_functions:` entry, the names and types of message variables, the `??LevelFun:` entries, and the value for the `??Response:` entry. The `??HOL_functions:` entry gives definitions of constants that are referred to in the rest of the IPSL specification. It

makes the HOL90 theory `string` a parent of the current theory to define the type `:string` and defines `source_level` as a function that maps character strings (presumably the names of persons or processes originating messages) to security levels. The `??LevelFun:` entry for port `f_out` assigns the level `unclassified` to data being sent out through `f_out`. The `??LevelFun:` entry for port `f_in` assigns the level of the source to data coming in through `f_in`. The `??Response:` value asserts that the filter will send a message out the port `f_out` if the message is from a source whose level is `unclassified`, but otherwise ignore it, and then call its top-level process to wait for the next message.

Translating this file by invoking the `ips12hol` translator on the top-level process

```
ips12hol simple_example
```

produces the files `filter.spec.sml`, `filter.goal.sml`, and `simple_example_globals.sml`, as well as goal and specification files for each of the other processes.

The file `simple_example_globals.sml` contains global definitions that are used by all the processes in a system. This file is used to create the global definitions theory with the following command:

```
rh01 <simple_example_globals.sml
```

The command `rh01` invokes a version of HOL90 that has the Romulus library preloaded. The HOL theory produced by this file is automatically made a parent of the HOL theories for each subcomponent when the top-level component is translated.<sup>1</sup> This means that, if the specifications of the subcomponent use definitions from the global definitions file, then you should always translate the top-level component in order to generate the correct goal and HOL specification files for each component.

We do not discuss the `filter.spec.sml` file here, but note that giving it as input to the HOL90 theorem prover with the UNIX command

```
rh01 < filter.spec.sml
```

produces the two files `filter.holsig` and `filter.thms`, which together implement a saved HOL90 version of the theory of the `filter` process.

---

<sup>1</sup>In the normal course of events you should complete the IPSL specification of the top-level process before starting work on the specifications of any of its subcomponents.

## 2.2.2 Proving the Filter Secure

It now remains only to prove that the filter is restrictive and satisfies the interface conditions and then to store this result in a form that can be recognized by the Romulus graphical interface.

We then run the HOL90 theorem prover with the UNIX command

```
rhof
```

and give it the file `filter.goal.sml` as input with the command

```
use "filter.goal.sml";
```

This file loads the HOL theory describing the `filter` process, defines a few names and constants, and then sets up the goal of proving `BNPSP_restrictiveness` for the `filter` process.

```
g('BNPSP_restrictive
  ^filterInPred
  ^filterOutPred
  (standard_dom)
  ^filterInLevel
  ^filterOutLevel
  ^filterInvocVal
  ^filterTop');
```

HOL responds by making this goal the top goal on the goal stack.

```
(--'BNPSP_restrictive filterInPred filterOutPred standard_dom
  filterInLevel
  filterOutLevel
  filterInvocVal
  filterTop'--)
```

```
=====
```

```
val it = () : unit
val it = () : unit
```

`BNPSP_restrictive` describes the restrictiveness condition and interface conditions for buffered, non-parameterized (i.e., memoryless) server processes. In the above, `filterInPred` and `filterOutPred` are functions that describe the level range conditions for input and output events (they are derived from

the filter's IPSL ??LevelRange: values), `standard_dom` defines the standard security levels in their standard order, `filterInLevel` and `filterOutLevel` are functions that assign levels to input and output events (they are derived from the filter's IPSL ??LevelFun: values), `filterInvocVal` is a function that assigns meanings to some of the symbols used in the PSL definition of the `filter` process, and `filterTop` denotes the full filter process.

The first step in the proof is to apply a standard Romulus tactic that expands the definition of `BNPSP_restrictive`, makes case splits on possible input events, and does rewrites to simplify away references to PSL. The command

```
e(BNPSP_restrictive_TAC);
```

reduces the goal to two subgoals

```
2 subgoals:
(--'standard_dom unclassified unclassified'--)
=====
  (--'source_level source = unclassified'--)
  (--'standard_dom (source_level source) unclassified'--)
  (--'standard_dom top_secret (source_level source)'--)

(--'standard_dom unclassified unclassified'--)
=====
  (--'source_level source = unclassified'--)
  (--'standard_dom (source_level source) unclassified'--)
  (--'standard_dom top_secret (source_level source)'--)
```

```
val it = () : unit
```

each of which says that the standard level unclassified dominates itself. This subgoal is easily proved by rewriting with the definition of the `standard_dom`.

```
e(REWRITE_TAC [definition "romlemmas" "standard_dom"]);
```

which proves first subgoal and gives the response

```
Goal proved.
|- standard_dom unclassified unclassified
```



```

Remaining subgoals:
(--'standard_dom unclassified unclassified'--)
=====
  (--'source_level source = unclassified'--)
  (--'standard_dom (source_level source) unclassified'--)
  (--'standard_dom top_secret (source_level source)'--)

```

```
val it = () : unit
```

Applying the same tactic again

```
e(REWRITE_TAC [definition "romlemmas" "standard_dom"]);
```

produces the following response

```

Goal proved.
|- standard_dom unclassified unclassified

Goal proved.
|- BNPS_restrictive filterInPred filterOutPred standard_dom
   filterInLevel
   filterOutLevel
   filterInvocVal
   filterTop

Top goal proved.
val it = () : unit

```

which shows that the second subgoal and consequently the original goal have been proved.

The final commands

```

save_top_thm("filter_BNPS_restrictive");
romrtheory("filter");
export_theory();
exit();

```

produce new versions of the `filter.holsig` and `filter.thms` files, containing the new theorem `filter_BNPS_restrictive`, and produces the `rtheory` file `filter.rth` for communicating this result back to the Romulus graphical interface.

As a final note, it is usually desirable to save the proof so that it can be redone at a later date. One way to do this is to rename the file `filter.goal.sml` to `filter.proof.sml`, so that it will not be overwritten

by any future use of the `ips12hol` translator, and then to edit the file `filter.proof.sml` to add each step of the proof. A `.goal` file contains comments to help you to do this. For longer proofs it is more convenient to edit the `.proof` file as you construct the proof rather than wait until the proof is finished. You can even copy text from the window in which you edit the proof file to the window in which you are running `rh1` to avoid typing everything twice.

## 2.3 A Manifestly Secure Process

The `unclassified_process` process is an example of an atomic process that handles data only at a single level and therefore is assumed by the flow analyzer to be manifestly secure. In fact, any process for which all its outputs are at higher levels than any of its inputs is assumed by the flow analyzer to be manifestly secure. All that needs to be checked for manifestly secure processes is that the assumptions made about them by the flow analyzer are correct. The condition that must be proved for such processes is that its output events are in the correct range and that if all its input events are in the specified range then the level of every output event will dominate the level of every input event. This *manifest security condition* depends only on the interface specified for the process, not on what the process does.

It is possible that you will be unable to prove the manifest security conditions for a process that was assumed by the flow analyzer to be manifestly secure. This may mean that the process must be trusted, in which case you must provide additional information about what the process does. The process must then be proved secure as described in the previous section.

### 2.3.1 Specifying the Unclassified Process

We prepare a formal specification of the `unclassified_process` component by first completing an IPSL specification for it, then translating the specification into a HOL90 specification. We start with the file `unclassi-`

fied\_process.ipsl produced by the save command.

```
??Process: unclassified_process

??OutPort: u_out
??MessageVar:
??LevelFun: unclassified
??LevelRange: unclassified unclassified

??InPort: u_in
??MessageVar:
??LevelFun:
??LevelRange: unclassified unclassified
??Response:
```

```
??EndProcess: unclassified_process
```

This file is a partial IPSL specification of the **unclassified\_process** process that contains everything that the graphical interface initially knows about this process.

We edit `unclassified_process.ipsl` to produce the following file:

```
??Process: unclassified_process
??HOL_functions:
new_parent "string";

??OutPort: u_out
??MessageVar: source:string
??MessageVar: data:string
??LevelFun: unclassified
??LevelRange: unclassified unclassified

??InPort: u_in
??MessageVar: source:string
??MessageVar: data:string
??LevelFun: unclassified
??LevelRange: unclassified unclassified
??Response:

??EndProcess: unclassified_process
```

The main additions here are the `??HOL_functions:` entry, the names and types of message variables, and the `??LevelFun:` entries. These are similar to those for the **filter** process. No `??Response:` entry is given for this process because, as long the manifest security conditions hold, the security of this

process does not depend on what it actually does. Leaving the `??Response:` entry blank is the signal to the translator to generate the simpler security condition that only checks the manifest security conditions.

Translating this file by invoking the `ips12hol` translator on the top-level component

```
ips12hol simple_example
```

produces the files `unclassified_process.spec.sml` and `unclassified_process.goal.sml`, as well as the `globals` file and goal and specification files for the other processes. Assuming that the `globals` theory has already been created, giving the `unclassified_process.spec.sml` file as input to the HOL90 theorem prover with the UNIX command

```
rhof < unclassified_process.spec.sml
```

produces the two files `unclassified_process.holsig` and `unclassified_process.thms`, which together implement a saved HOL90 version of the theory of the `unclassified_process` process.

### 2.3.2 Proving the Unclassified Process

It now remains only to prove that the `unclassified_process` satisfies its manifest security conditions and then to store this result in a form that can be recognized by the Romulus graphical interface.

We then run the HOL90 theorem prover with the UNIX command

```
rhof
```

and give it the file `unclassified_process.goal.sml` as input with the command

```
use "unclassified_process.goal.sml";
```

This file loads the HOL theory describing the `unclassified_process` process, defines a few names and constants, and then sets up the goal of proving the manifest security conditions.

```
g('!outev inev.
  ((~unclassified_processInPred inev) ==>
    ((standard_dom) (~unclassified_processOutLevel outev)
      (~unclassified_processInLevel inev))) /\
  (~unclassified_processOutPred outev)');
```

HOL responds by making this goal the top goal on the goal stack.

```
(--'!outev inev.
  (unclassified_processInPred inev ==>
    standard_dom (unclassified_processOutLevel outev)
      (unclassified_processInLevel inev)) /\
    unclassified_processOutPred outev'--)
=====
```

```
val it = () : unit
```

In the above, `unclassified_processInPred` and `unclassified_processOutPred` are the functions that describe the level range conditions for input and output events (they are derived from the `unclassified_process`'s `IPSL ??LevelRange: values`), `standard_dom` defines the standard security levels in their standard order, and `unclassified_processInLevel` and `unclassified_processOutLevel` are the functions that assign levels to input and output events (they are derived from the `unclassified_process`'s `IPSL ??LevelFun: values`).

The first step in the proof is to apply the standard Romulus tactic for this goal.

The command

```
e(ManifestlySecure_TAC);
```

reduces the goal to one subgoal

```
1 subgoal:
  (--'standard_dom unclassified unclassified'--)
=====
```

```
val it = () : unit
```

which says that the standard level `unclassified` dominates itself. This subgoal is easily proved by rewriting with the definition of the `standard_dom`.

```
e(REWRITE_TAC [definition "romlemmas" "standard_dom"]);
```

The application of this tactic produces the following response

```

Goal proved.
|- standard_dom unclassified unclassified

Goal proved.
|- !outev inev.
  (unclassified_processInPred inev ==>
   standard_dom (unclassified_processOutLevel outev)
   (unclassified_processInLevel inev)) /\
   unclassified_processOutPred outev

Top goal proved.
val it = () : unit

```

which shows that the subgoal and consequently the original goal have been proved.

The final commands

```

save_top_thm("unclassified_process_ManifestlySecure");
romrtheory("unclassified_process");
export_theory();
exit();

```

produce new versions of the `unclassified_process.holsig` and `unclassified_process.thms` files containing the new theorem `unclassified_process_ManifestlySecure`, and produce the `rtheory` file `unclassified_process.rth` for communicating this result back to the Romulus graphical interface.

Again, it is usually desirable to rename the file `unclassified_process.goal.sml` to `unclassified_process.proof.sml` and edit it to contain the proof.

## 2.4 A Composite Process

The `simple_example` process is an example of a composite process, a process constructed by connecting together other processes. Romulus proves that such processes are secure by proving each of their subprocesses restrictive and then proving that the pieces are properly connected. The restrictiveness of the composite process then follows from the hookup property of restrictiveness. This section describes how these conditions are proved using the `simple_example` process as an example.

### 2.4.1 Specifying the Simple Example

We prepare a formal specification of the top-level process `simple_example` component by first completing an IPSL specification for it, then translating the specification into a HOL90 specification.

We start with the file `simple_example.ipsl` produced by the `save` command.

```
??Process: simple_example

??OutPort: s_out
??MessageVar:
??LevelFun:
??LevelRange: unclassified unclassified

??InPort: s_in
??MessageVar:
??LevelFun:
??LevelRange: unclassified top_secret

??ProcessInFile: filter
??ProcessInFile: unclassified_process

??Connection: c2p1 p1
??Connection: p2 c1p2
??Connection: c1p1 c2p2

??EndProcess: simple_example
```

This file is a partial IPSL specification of the `simple_example` process that contains everything that the graphical interface initially knows about this process. The `??ProcessInFile:` declarations define `simple_example` in terms of the processes `filter` and `unclassified_process` and `??Connection:` entries describe connections between ports using tree addresses.

We edit `simple_example.ipsl` to produce the following file:

```
??Process: simple_example
??HOL_functions: new_parent "string";

??OutPort: s_out
??MessageVar: source:string
??MessageVar: data:string
??LevelFun: unclassified
```

```
??LevelRange: unclassified unclassified
```

```
??InPort: s_in
```

```
??MessageVar: source:string
```

```
??MessageVar: data:string
```

```
??LevelFun: source_level source
```

```
??LevelRange: unclassified top_secret
```

```
??ProcessInFile: filter
```

```
??ProcessInFile: unclassified_process
```

```
??Connection: c2p1 p1
```

```
??Connection: p2 c1p2
```

```
??Connection: c1p1 c2p2
```

```
??EndProcess: simple_example
```

The main additions here are the `??HOL_functions`: entry, the names and types of message variables, and the `??LevelFun`: entries. These are similar to those for the `filter` process.

Translating this file with the `ips12hol` translator using the UNIX command

```
ips12hol simple_example
```

produces the files `simple_example.spec.sml`, `simple_example.goal.sml`, and `simple_example_globals.sml`, as well as goal and specification files for the other processes. Assuming that the global definitions theory has already been created, giving the `simple_example.spec.sml` file as input to the HOL90 theorem prover with the UNIX command

```
rh01 < simple_example.spec.sml
```

produces the two files `simple_example.holsig` and `simple_example.thms`, which together implement a saved HOL90 version of the theory of the `simple_example` process.

## 2.4.2 Proving the Simple Example

It now remains only to prove that the parts of `simple_example` are properly connected and then to store this result in a form that can be recognized by the Romulus graphical interface. Two things must be proved for a connection



in order for it to be properly connected. Consider a connection from one port to another, that is, events are transmitted from the first port to the second port. First, the level range of the first port must be a subrange of the level range of the second port. Second, the same level must be assigned to an event leaving the first port as is assigned to the same event entering the second port.

We run the HOL90 theorem prover with the UNIX command

```
rh01
```

and give it the file `simple_example.goal.sml` as input with the command

```
use "simple_example.goal.sml";
```

This file loads the HOL theory describing the `simple_example` process, defines a few names and constants, and then sets up the goal of proving that the connections are properly made.

```
g('
  (! (source:string) (data:string).
    (unclassified_processOutPred (u_out (source:string) (data:string)))
    ==>
      ((simple_exampleOutPred (s_out (source:string) (data:string)))
       /\ ((unclassified) = (unclassified))))
  /\
  (! (source:string) (data:string).
    (simple_exampleInPred (s_in (source:string) (data:string)))
    ==>
      ((filterInPred (f_in (source:string) (data:string)))
       /\ ((source_level source) = (source_level source))))
  /\
  (! (source:string) (data:string).
    (filterOutPred (f_out (source:string) (data:string)))
    ==>
      ((unclassified_processInPred (u_in (source:string) (data:string)))
       /\ ((unclassified) = (unclassified))))
  ');
```

HOL responds by making this goal the top goal on the goal stack.

```
(--'(!source data.
  unclassified_processOutPred (u_out source data) ==>.
  simple_exampleOutPred (s_out source data) /\
```

```

      (unclassified = unclassified)) /\
    (!source data.
      simple_exampleInPred (s_in source data) ==>
      filterInPred (f_in source data) /\
      (source_level source = source_level source)) /\
    (!source data.
      filterOutPred (f_out source data) ==>
      unclassified_processInPred (u_in source data) /\
      (unclassified = unclassified))'---)
=====

```

```

val it = () : unit
val it = () : unit

```

In the above, `simple_exampleInPred` and `simple_exampleOutPred` are the functions that describe the level range conditions for input and output events (they are derived from the `simple_example`'s `IPSL ??LevelRange` values).

The first step in the proof is to apply the standard Romulus tactic for this goal. The command

```
e(HookupValid_TAC);
```

produces the following response

```

Goal proved.
|- (!source data.
  unclassified_processOutPred (u_out source data) ==>
  simple_exampleOutPred (s_out source data) /\
  (unclassified = unclassified)) /\
  (!source data.
    simple_exampleInPred (s_in source data) ==>
    filterInPred (f_in source data) /\
    (source_level source = source_level source)) /\
  (!source data.
    filterOutPred (f_out source data) ==>
    unclassified_processInPred (u_in source data) /\
    (unclassified = unclassified))

```

```

Top goal proved.
val it = () : unit

```

which shows that the original goal has been proved.

The final commands

```
save_top_thm("simple_example_HookupValid");
romrtheory("simple_example");
export_theory();
exit();
```

produce new versions of the `simple_example.holsig` and `simple_example.thms` files, containing the new theorem `simple_example_HookupValid`, and produces the `rtheory` file `simple_example.rth` for communicating this result back to the Romulus graphical interface.

Again, it is usually desirable to rename the file `simple_example.goal.sml` to `simple_example.proof.sml` and edit it to contain the proof.

## 2.5 Confirming Proof Completion

To confirm the completion of all required security proofs, run the Romulus graphical interface on the saved system model with the UNIX command

```
romulus -initial=simple_example
```

Selecting the `check` command in the `Component operations` row and then clicking on the `filter` component with the left mouse button will cause the graphical interface to read the file `filter.rth` to confirm that the `filter` process has been proven secure. If everything checks out, two asterisks will appear in the lower-right corner of the component's box, indicating that it has been proved secure.

Selecting the `check` command and clicking on the `unclassified_process` component with the left mouse button will cause the graphical interface to read the file `unclassified_process.rth` to confirm that the `unclassified_process`'s manifest security conditions have been proved. If everything checks out, two asterisks will appear in the lower-right corner of the component's box, indicating that its manifest security conditions have been proved.

Selecting the `check` command and then clicking on the `simple_example` component with the left mouse button will cause the graphical interface to read the file `simple_example.rth` to confirm that the `simple_example` process's have been properly connected. If everything checks out, then two asterisks will appear in the lower-right corner of the component's box, indicating that it has been proved secure.

The following chapters discuss the graphical interface, the Romulus theories of processes and process security, and IPSL in detail.

---

## Chapter 3

# The Romulus Graphical Interface

The Romulus graphical interface is a tool for partially creating and displaying formal specifications of multilevel systems and determining the extent to which they are restrictive, which means having a strong nondisclosure security property. The graphical interface, together with the translator `ips12hol` that is supplied with it, allows the user to do the following:

- create graphical representations of communicating processes and the information flow connections between them;
- save and restore these graphical representations;
- produce PostScript files giving hard-copy images of these graphical representations;
- attach level ranges to connections between processes that bound the security levels of the events passing through these connections;
- determine, by flow analysis, assuming that the attached level ranges on connections are correct, which atomic (i.e., having no subprocesses) subprocesses must be restrictive in order to guarantee that the full system is restrictive;
- automatically generate partial Interface Process Specification Language (IPSL) specifications of processes;

- translate user-completed IPSL specification files into pairs of files that give formal HOL90 specifications of these processes, formal HOL90 statements of the properties that must be proved for the processes, a suggested major initial step for proving these properties, and a call to the Romulus HOL90 utility `romrtheory`, which produces files that communicate the results proved to the Romulus graphical interface; and
- confirm which atomic subprocesses have been proved restrictive, which processes are manifestly secure, and which composite processes are properly connected.

The graphical interface also uses the files produced by `romrtheory`, called *rtheory* files, to identify the security levels that it recognizes.

The Romulus graphical interface and the `ips12hol` translator are used in conjunction with a HOL90 prover and files that specify Romulus-specific theories of processes and process security, Romulus-specific utilities for declaring event and state parameter types (including records), Romulus-specific tactics for proving restrictiveness, manifest security conditions, and proper connectivity, and the utility `romrtheory`. All these theories, utilities, tactics, and `romrtheory` are given in the Romulus HOL90 library, described in detail in Appendix B.

The cores of Romulus atomic process specifications, both in IPSL specifications and in the corresponding full HOL90 specifications, are given in terms of a concrete recursive type PSL (for “Process Specification Language”) whose elements are interpreted as computer programs giving the remaining actions in the process. Several of the basic constructs in PSL—`Skip`, `If`, `Send`, `Receive`, and the followed-by operator `;;`—are modeled after analogous constructs in CSP [5].

Although PSL is used to specify processes, the standard Romulus tactics for showing restrictiveness typically remove all references to PSL in the remaining subgoals to be proved, leaving only relatively simple statements about the security levels of messages and the characteristics of process state parameters.

This chapter gives basic information on the Romulus graphical interface’s capabilities, the `ips12hol` translator’s capabilities, and how they can be used. The `ips12hol` translator is described in section 3.5.8 on the spec

command. The `ips12hol` translator provides the same functionality as the `spec` command but can be used independently of the graphical interface. Chapter 5 gives additional information about IPSL, PSL, and `rtheory` files.

For the remainder of this chapter, “Romulus” will refer to the Romulus graphical interface.

### 3.1 Basic Concepts and Terminology

The basic objects in Romulus are *components* and *ports*. A component is an abstract representation of a process. Components in Romulus are arranged in a tree with the main component being studied, typically a process corresponding to a full system, at the top and the subcomponents of a component occurring below that component in the tree. A component’s immediate subcomponents are called its *children*; its general subcomponents are called its *descendants*. The component that has another component as an immediate subcomponent is called that subcomponent’s *parent*; the general components that have that subcomponent as a descendant are called its *ancestors*.

A port, which can be either an input port or an output port, is an abstract representation of a connection through which data enters or leaves a process. A port that is an entry or exit point for data into or from a component is said to be *owned* by that component, and that component is called the port’s *owner*. A connection between an output port and an input port indicates that the data that flows from the output port flows into the input port. A connection between two input ports or two output ports can be made only when one port’s owner is the parent of the other port’s owner. Such a connection indicates either that the data that flows into the parent component’s input port flows through to enter the child component’s input port, or that the data that flows from the parent component’s output port comes from the child component’s output port. See Figure 3.1 for examples. In this figure, input ports are pictured as disks with holes in their centers, output ports are pictured as solid diamonds, and connections between ports are pictured as arrows pointing in the direction of data flow.

The following information associated with a component can be set using the graphical interface: its name; its status identifying it as proven secure, assumed to be secure, or possibly insecure; a list of the ports owned by the component; and a list of the component’s children.

The following information associated with a port can be set using the graphical interface: its name; possibly a lower bound on the possible security levels of events passing through the port; possibly an upper bound on the possible security levels of events passing through the port; and the port's type, either input or output.

In addition, both components and ports have associated information that controls how and where icons representing them are displayed on the user interface's *canvas*.

## 3.2 General Interface Principles

The Romulus program uses several user-preference parameters that can be set in Romulus defaults files, environment variables, or in command-line arguments. These parameters are described in Appendix A. In addition, the code sets reasonable default values for these parameters and accepts standard X command-line arguments, particularly *-geometry*, so the user can set the initial size and shape of the Romulus window. The Romulus window is not currently designed to be resizable, though, so the effect of resizing it is unpredictable.

If it is run without setting the parameter *initial* to an appropriate basename, Romulus comes up describing an empty component. If it is run with the parameter *initial* set to a basename, and if the file with this basename and the extension *.rom* contains a valid saved description of a component—a saved description of the form produced by Romulus's top-level *save* command described below—Romulus comes up describing the component given in that file.

Romulus comes up in an ordinary window managed by the currently active window manager. Figure 3.1 shows the Romulus graphical interface running in an X11 window and displaying a model of a token ring station. Romulus divides this window into the following five areas, only four of which are initially visible: *command buttons*, a *message window*, the *canvas window*, *inert background areas*, and *text-entry windows*. The text-entry windows are not initially visible; they appear only when the commands with which they are associated are selected. At most one text-entry window is visible at a time. In addition, the top-level *modify* command and the port and component display commands are associated with special-purpose windows



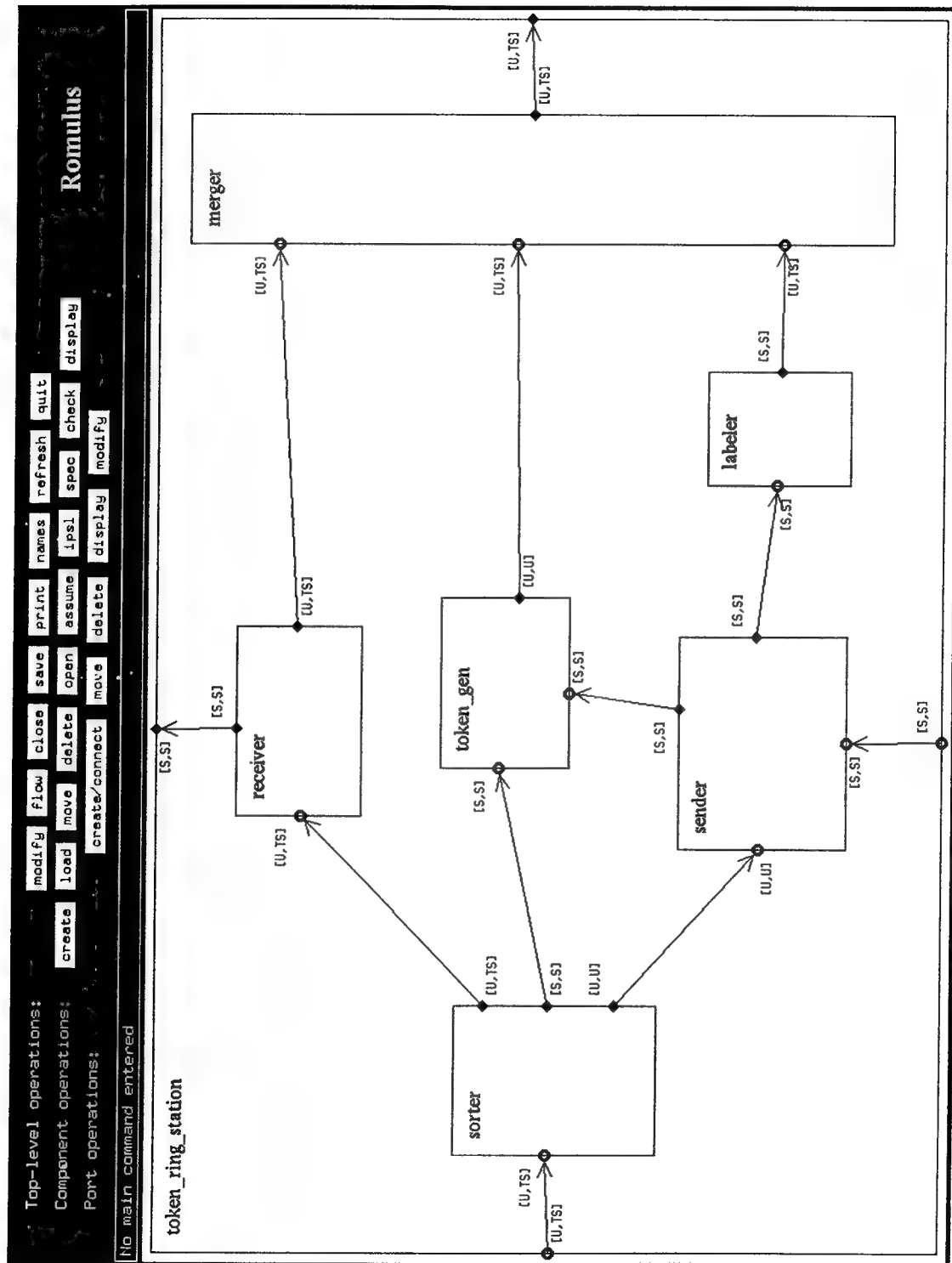


Figure 3.1: Romulus window displaying a token ring station

that appear over the canvas.

### 3.2.1 Command Buttons

The top of the interface window contains three rows of buttons for top-level, component, and port operations respectively. Moving the mouse cursor onto one of these buttons *enters* it, and then clicking the left mouse button *selects* it. When a command button is entered, a short message describing the associated command appears in the message window; when a command button is selected, a message describing necessary further input, if any, appears in the message window. At most one command can be selected at any time. If a button is selected, the colors of the button and its label are reversed. Selecting a different command button automatically deselects any previously selected command button. Selecting the currently selected command button again deselects it.

If a command button is selected, that command is said to be *active*. Some active commands (e.g., quit) are carried out as soon as they are activated; others (e.g., save) are carried out after the user enters necessary text in a text-entry window; others (e.g., delete) are carried out each time the user makes appropriate mouse clicks and/or movements on the canvas; others (e.g., load) are carried out only after the user enters necessary text and are then carried out each time the user makes appropriate mouse clicks and/or movements on the canvas; and still others (e.g., modify) are carried out each time the user makes appropriate mouse clicks and modifies text in an edit window that appears over the canvas.

All commands can be deselected by selecting them again or by clicking on the right mouse button with the mouse cursor on the canvas. Commands that can be carried out immediately are deselected automatically. Commands that require additional user input stay active until that input is provided or until the user deselects them.

### 3.2.2 Text-Entry Windows

There are two different text-entry windows, at most one of which is visible at a time. These windows appear only when a command requiring the user to enter text is active. When one of these windows appears, it occupies the position otherwise occupied by the **Romulus** logo in the upper-right corner

of the Romulus window. The other kind of text-entry window appears where you make a selection on the canvas.

Each of these windows contains two different types of active subareas: a *text-editing window* and one or more *text-use subcommand* buttons.

The text-editing window is a short, empty rectangle used for entering character strings. When such a window appears, all keyboard output is directed to it whenever the mouse cursor is in the Romulus window. It supports all the Emacs-style editing capabilities of an Athena Widget Set "Text" widget. The editing capabilities that a user will typically need are the following: backspace, which deletes the last character typed; control-b, which moves the cursor to the left; and control-f, which moves the cursor to the right. If the user enters a line-feed or carriage-return character in this window, Romulus treats it as an error.

The text-use subcommand buttons are entered and selected in the same way as the command buttons, a selected text-use subcommand button is similarly highlighted, and similar descriptive messages appear in the message window. Like the command buttons, only one subcommand button can be selected, and reselecting a selected subcommand button deselects it. (There is one exception: the `confirm` button serves no function for the `load` command and cannot be selected.) These buttons identify how the entered text will be used or confirm that entry of the text has been completed.

### 3.2.3 Message Window

The message window is a short, wide window right below the command buttons. It always contains a message reminding the user of the current command state and the user's options for continuing the command.

When a command or text-use subcommand button is entered, this window contains a description of the command or text-use. If a command button is selected, and if the command requires additional user input before it can be carried out, the window contains a message describing that input. It prompts for text input if it is necessary. For commands that require canvas mouse clicks and/or movements, the window contains descriptions of the functions of the different mouse buttons and of the effects of clicking or holding them. For commands that require both text-window text entry and mouse clicks, after the command is selected the window describes the possible mouse clicks and prompts for the necessary text. The message window changes to give

descriptive messages for the different text-use possibilities after the text-entry subcommand buttons are entered, but after a text-use subcommand button is selected the window changes back to contain the description of the effects of possible mouse clicks.

### 3.2.4 Canvas Window

The canvas is a large white rectangle that occupies most of the Romulus window. Components are drawn on the canvas as rectangles with their names, if they have them, in their upper-left corners. An unnamed component has a *tree address* in the upper-left corner instead of a component name; tree addresses are in parentheses, which distinguishes them from component names. In a tree address, **T** denotes the top-level component, and **c1**, **c2**, **c3**, and so on denote a component's first, second, third, and so on child component. The tree address **Tc2c3**, for example, denotes the third child component of the second child component of the top-level component. Child components are numbered in the order in which they were created. Components are drawn with zero, one, or two asterisks in their lower-right corners, showing that they are possibly insecure, assumed secure, or proved secure respectively.

Input ports are drawn as disks having holes in their centers and output ports are drawn as solid diamonds. Connections between ports are represented as arrows showing the directions of data flows through these connections. Romulus also shows ports whose types have not yet been determined, during use of the `create/connect` command, as solid squares. The range of security levels of messages through each port is shown as an interval in brackets using the abbreviations given in the Romulus `abbreviations` parameter. These abbreviations are expected to be short, one or two characters long, but Romulus will use them even if they are not short and will use full names of security levels for which no abbreviations are given. Romulus uses an underscore to denote an upper or lower security-level limit that is not present, so the level range `[_,_]` indicates that the associated port has no security-level limit information.

The canvas always contains one component called the *open* component whose rectangle occupies almost all of the canvas; this rectangle contains the rectangles of all the open component's child components if it has any. All Romulus component operations affect only the open component and its children; all Romulus port operations affect only ports owned by the open

component and its children. The `open` and `close` commands, described below, allow the user to change which component is open.

The user *selects* a child component of the open component by moving the mouse cursor into the rectangle for that component and clicking the left mouse button. The user selects the open component by moving the mouse cursor to a point inside its rectangle, but outside the rectangles of its children, and clicking the left mouse button.

The user selects a port by moving the mouse cursor onto the icon for that port and clicking either the left or middle mouse button, depending on the currently active command and the desired results. Both input and output port icons are actually drawn in same-sized squares slightly larger than their visible boundaries, and the user can select a port by clicking on the left or middle mouse button with the mouse cursor at any point in this square.

For the port `delete` and port `modify` commands, the user can also select the connection between two ports. The user selects a connection by moving the mouse cursor to within one port-width of the center line for the arrow showing this connection, then clicking either the left or middle mouse button, depending on the desired effect.

Commands that display information do so by creating windows that overwrite part of the canvas. Romulus's error-message windows also overwrite part of the canvas. Information displays and error messages stay up until the user clicks a mouse button with the mouse cursor either on the canvas or inside a command button, in which case the click removes the information display or error message, redraws the canvas, and has no further effect.

When the mouse cursor is on the canvas, the functions of the left and middle mouse buttons vary with whichever command is active, as described below, but clicking the right mouse button will always deselect the active command unless it causes an information display or error message to be removed.

### 3.2.5 Inert Background Areas

The areas on the Romulus window that are not in the command buttons, the canvas, or any currently displayed text-entry window are inert. Mouse buttons with the mouse cursor in one of these areas have no effect. The interface is designed so that, in general, inert areas are black and other

areas are white—selected command buttons with their colors reversed are an exception.

### 3.3 Command Levels

As noted above, the Romulus commands are arranged in three rows, each of which roughly corresponds to a level of abstraction in viewing or analyzing the system being studied. Commands in the second two rows, which will typically be used most often, are arranged so that the expected sequence of commands will move from left to right.

Commands in the first row, the *top-level* operations, refer to the global state of the interface, to the component that is currently open, or to both ports and components. The top-level commands are *modify*, *flow*, *close*, *save*, *print*, *names*, *refresh*, and *quit*. Of these commands, only *save*, which saves a description of the open component, and *modify*, which allows the user to modify text strings associated with ports and/or components, require additional user input after they are selected and before they are executed. The *save* command causes a text-entry window for entering the main save file's basename to appear. After the user selects a port or component to modify with a mouse click, the *modify* command causes an edit window to appear. All of these commands except *modify* and *close* deselect themselves when they are executed.

Commands in the second row, the *component* operations, mainly refer to the children of the open component. The component commands are *create*, *load*, *move*, *delete*, *open*, *assume*, *ipsl*, *spec*, *check*, and *display*. Five of them, *assume*, *ipsl*, *spec*, *check*, and *display*, can also act on the open component itself. These five commands have this property mainly because they might need to be applied to the top component being studied, which has no parent component; allowing these commands to apply to the open component also frequently avoids having to change which component is open. The other five commands in this row are not meaningful, or are not appropriate, for the open component.

All the component operations require mouse actions on the canvas before they are executed, but all of them remain active until the user deselects them. In addition, the *load* command requires the user to enter the basename of a file that contains a saved description of a component.

Commands in the third row, the *port* operations, refer to the ports owned by the open component and its children and to the connections between these ports. The port commands are `create/connect`, `move`, `delete`, `display`, and `modify`. All of these commands require mouse actions on the canvas before they are executed. One of them, `modify`, also requires the user to enter a text string and identify it as being the new lower or upper bound on the security levels of information passing through the port for any port selected to be modified. The `modify` command also allows the user to select a connection between two ports and in this case modifies both of the ports at the ends of this connection.

## 3.4 Top-Level Commands

This section describes the first row of the command buttons, the top-level commands. These do the following: make changes to both components and ports; analyze, save, and print the currently open component; or change the state of Romulus to control what is displayed.

### 3.4.1 `modify`

This command allows the user to modify the text strings give the name or security level limits associated with any displayed component or port. The user selects a component or port by clicking on its icon with the mouse, and the command then causes an edit window to appear at the point selected. This window displays the strings that can be modified and labels identifying these strings.

The user modifies these strings using the Emacs-like Athena Text Widget editing commands. When editing modifications are completed, the user can make these modifications to the affected component or port by either keyboard input or additional mouse clicks. The user can also discard any modifications with either keyboard input or mouse clicks. Any modification of the identifying labels is treated as an error and causes all modifications to be discarded.

By default, the keyboard sequences `control-s` and `control-c` carry out or cancel pending modifications, respectively. The user can change which keyboard sequences have these interpretations by adjusting the Romulus

translationstable parameter.

Selecting a component or port with the left mouse button first carries out any pending modifications of a previously selected component or port, then causes an edit window to appear allowing the user to make pending modifications to all the text strings associated with the selected component or port. Selecting any point on the canvas with the middle mouse button cancels any pending modifications but leaves the modify command active. Selecting any point on the canvas with the right mouse button carries out any pending modifications and deselects the modify command. It is expected that the user will make a sequence of selections with the left mouse button, make pending modifications after each selection, then confirm the last modification and end the command with the right mouse button.

### 3.4.2 flow

This command either invokes the flow analyzer on the open component or removes the display of a previously identified, potentially insecure data flow.

If the open component has already been proved or assumed secure, invoking the flow analyzer has no effect. If the open component has not been proved or assumed secure, the flow analyzer examines all possible data flows inside the open component. It assumes, for every component *C*, that data entering any input port owned by *C* can flow to any output port owned by *C*—unless that input port is connected to an input port owned by one of *C*'s child components, in which case the data can flow only to that child. If the lower bound on a port has not been specified, the flow analyzer assumes that the lower bound for that port is system low. Likewise, if the upper bound on a port has not been specified, the flow analyzer assumes that the upper bound for that port is system high. If all these data flows are manifestly secure, meaning that the highest level events passing through input ports have security levels that are no higher than the lowest level events passing through output ports, then a message is displayed that says the open component is secure if the hookups between its components are valid. If the flow analyzer finds a data flow that is not manifestly secure, hence possibly insecure, it shows this data flow by drawing all connections in it with bold arrows, drawing all port icons for ports in it with boxes around them, and drawing all components that own two of these ports with bold boundaries. For ports with no level ranges set, the flow analyzer assumes that information of any



level can pass through the port.

When a possible insecure data flow is displayed, the user can still use all commands except **flow** as they are normally used. In particular, the user can invoke the **open** and **close** commands to view all of the flow if parts of it are hidden inside child components. If the user invokes a command that changes the conditions under which the possible insecure flow was detected (e.g., commands that delete ports or connections in the flow), the flow is automatically no longer displayed.

If a possible insecure data flow is displayed and involves at least one port, connection, or component on the canvas, selecting **flow** causes this data flow to no longer be displayed. If the data flow does not involve a port, connection, or component showing on the canvas for the open component—which can happen if the user invokes the flow analyzer on a possibly insecure component, then opens one of that component's children that is not involved in the possibly insecure data flow—the **flow** command causes an error message to appear and has no further effect.

### 3.4.3 close

This command changes the open component to the parent of the previously open component. If the open component is already the full system, **close** causes an error message to appear and has no further effect.

### 3.4.4 save

This command saves a textual representation of the open component, its subcomponents, their ports, and the connections between these ports in text files. Graphical information that describes the sizes and positions of the components and ports of the open component is saved in a file whose basename is provided by the user and has the extension **.rom**. The **save** command causes a text-entry window to appear in which the user enters the basename of the file; it automatically adds the extension **.rom**. In addition to the **.rom** file, the **save** command produces an **.ips1** file for the open component and for each of its subcomponents that contains information about names of components and ports, level ranges, connections between ports, etc.

The **save** is actually performed when the user selects the **confirm** sub-command in this window. The saved version of the open component does not

contain connections to any port outside the open component. Saved components can be restored with the `load` command or by giving the basenames of their `.rom` files as initial parameters. If an `.ips1` file for a component exists, Romulus renames the old `.ips1` file before the new `.ips1` file for the component is written. The first backup `.ips1` file is given extension `.old1.ips1`; subsequent backup files are given the extension `.oldn.ips1`, where *n* is one greater than the largest previously existing backup file.

The `save` command puts all information that it knows about a component into a component's `.ips1` file. This includes information that may have been read from an `.ips1` file for the component but cannot be modified or displayed by the graphical interface. This guarantees that, if you edit an `.ips1` file, the information you add to the `.ips1` file will be not be lost by loading the component into the graphical interface and saving it again.

We strongly recommend *always* invoking the `save` command from the top-level component. If you invoke `save` from a lower-level component and then exit Romulous, only part of your model will be saved.

### 3.4.5 `print`

This command produces a PostScript file giving the current contents of the canvas display. If the currently open component is named `name`, the command will produce a file with basename `name` and extension `.ps`. If the currently open component is not named, it will produce a file whose base-name is the currently open component's tree address with the extension `.ps`. The PostScript file can be printed on any PostScript printer in the usual way.

### 3.4.6 `names`

This command toggles whether or not port names, or port tree addresses for ports that are unnamed, are displayed for the ports appearing on the canvas. A port tree address consists of the tree address of the port's owner followed by `p1`, `p2`, `p3`, and so on to denote the first, second, third, and so on of this owner's ports. The tree address `Tc2c3p1`, for example, denotes the first port on the third child of the second child of the top-level component. Ports are numbered in the order they were created. Romulus uses a simple algorithm to place these names where they are reasonably unlikely to overlap with other parts of the canvas display, but overlaps are still possible.

### **3.4.7 refresh**

This command redraws the canvas. Since redrawing happens automatically in all ordinary situations in which it is required, using `refresh` is seldom necessary.

### **3.4.8 quit**

This command causes Romulus to exit.

## **3.5 Component Commands**

This section describes the second row of the command buttons, the component commands. These modify or examine the currently open component or one of its children.

### **3.5.1 create**

This command creates new child components of the open component. Pressing down the left mouse button determines the position of the upper-left corner of a new component, and moving the mouse cursor with the left mouse button held down changes the position of this new component's lower-right corner. The canvas display shows the rectangle that is currently the boundary of the new component. Releasing the left mouse button causes the new component to be created. When a new component is created, its ancestors' statuses are reset to "possibly insecure".

The command causes an error message to appear and has no further effect if the user inputs a component boundary that overlaps an existing child component of the open component, falls outside the boundary of the open component, or is too small to contain a name and ports.

### **3.5.2 load**

This command allows the user to load a previously saved component. It causes a text-entry window to appear in which the user specifies the base-name of the `.rom` file for the component; the command automatically adds the extension `.rom`. After the user enters such a basename, `load` makes

a copy of the saved component in the corresponding file into a new child component of the open component, doing so each time the user specifies a new child's location and dimensions. Pressing the left mouse button gives the position of the new child's upper-left corner and releasing the left mouse button, possibly after holding it and moving the mouse, gives the position of the child's lower-right corner. The canvas display shows the rectangle this process will determine while the left mouse button is being held and the mouse is being moved. If the dimensions of the new child are too small to contain a name and ports, `load` ignores the selected position of the lower-right corner and makes the new child's dimensions the same as those of the saved component. When it creates a new child component, `load` resets all that component's ancestors' statuses to "possibly insecure".

The command causes an error message to appear and has no further effect if the user does not enter a basename, if the associated file is not found, if the file does not contain a valid description of a component, if a new child component location overlaps the location of another child component, or if a new child component includes points not inside the open component.

The `load` command reads all the information in an `.ips1` file and stores it internally, even though some of this information cannot be modified or displayed by the graphical interface. However, all this information is written to `.ips1` files created by the `save` command. This guarantees that, if you edit an `.ips1` file, the information you add to the `.ips1` file will be not be lost by loading the component into the graphical interface and saving it again.

### 3.5.3 `move`

This command allows the user to move and/or resize child components of the open component. Pressing the left mouse button with the mouse cursor inside a child component selects that component as the one to be moved. Pressing the middle mouse button then gives the location to which that component's upper-left corner is moved. If the user holds down the middle mouse button and moves the mouse cursor, the cursor's position becomes the new position of the moved component's lower-right corner. The canvas display shows the rectangle that will be the new boundary of the moved component. Releasing the middle mouse button then moves and/or resizes the selected component.

If the new rectangle is too small to contain a name and ports, the com-

mand leaves the selected component's size and shape unchanged and simply makes the point determined when the middle mouse button was pressed the new location of the moved component's upper-left corner. In particular, clicking the middle mouse button simply moves the selected component.

The command causes an error window to appear and has no further effect if a new location is given before a component to be moved is selected, if the new component location overlaps the location of a child component other than the one being moved, or if the new location includes points not inside the open component.

#### **3.5.4 delete**

This command allows the user to delete child components of the open component. The user selects a child to be deleted with the left mouse button. The selected component, all its subcomponents, and all their ports are deleted. The command resets the statuses of all a deleted component's ancestors to "possibly insecure". If the deleted component was involved in a displayed possibly insecure data flow, it clears the display of this flow.

#### **3.5.5 open**

This command makes a child component of the open component into the new open component. The user selects this child component by clicking the left mouse button.

#### **3.5.6 assume**

This command allows the user to assume that a component is secure or possibly insecure, even if it was earlier proved secure. (This command is typically used in conjunction with flow analysis to see what potential insecure data flows are removed or created by changes in the security status of particular components.) The user selects either the open component or one of its children with either the left or middle mouse button. Selecting a component with the left mouse button causes its status to be reset to "assumed secure", and selecting it with the right mouse button causes its status to be reset to "possibly insecure". Assuming that a component is possibly insecure resets the statuses of all of its ancestors to "possibly insecure". If the component

whose assumed security status is changed was involved in a displayed possibly insecure data flow, the command clears the display of this flow.

### 3.5.7 ipsl

This command creates files containing a IPSL specifications of the selected component and its subcomponents. This command creates .ipsl files in exactly the same way as the `save` command. The only differences are that the `ipsl` command does not create a .rom file and the `ipsl` command can be used to create an .ipsl file for any currently displayed component whereas the `save` command always applies to the the open component.

If the component is named, the basename of the file the command produces is the component's name or the component's name suffixed with its tree address to make it unique, and otherwise the basename is the component's tree address. In all cases, the suffix of this file is .ipsl. Backup .ipsl files are made using the same technique as for the `save` command.

The `save` command is the preferred method for creating .ipsl files.

### 3.5.8 spec

This command translates the IPSL specification information associated with the selected component to produce two files, a specification file containing a HOL90 specification of the process associated with the selected component and a goal file that sets up the appropriate goal to be proved for the process. Comments are included in the goal file that suggest the first step of the proof and actions to be followed after the proof. If the selected component is a composite process then each of its subcomponents is also translated.

The HOL90 specification file produced by the translator removes any earlier versions of the process's theory, loads the Romulus library, and creates a new theory of the process in the environment determined by this library. It defines input events and output events for the process, defines functions assigning security levels to input and output events, and, for an atomic process, defines the process itself as a PSL object. For parameterized processes, it also defines the initial value of the process's state parameter, the invariant satisfied by this state parameter, and the projection function whose value on a security level and state parameter is the possibly sanitized state parameter giving all (but only) the process's behavior observable at that level.

The HOL90 goal file produced by the translator loads the Romulus library and the theory produced by the process's HOL90 specification file, sets the appropriate goal for the process, and gives (in a comment) the probable best first step in a proof of this goal. It also gives (again in a comment) the final lines that will save the resulting theorem, load `romrtheory`, and call `romrtheory` to produce an `rtheory` file for communicating the result back to Romulus.

The translation fails if it is unable to open the specification or goal files, if the component has missing or extraneous attributes (if it specifies a state parameter it must also specify a projection function, for example), if the component does not have both input and output ports, or if some port does not specify the names and types of elements in messages through the port.

If the component is named, the basename of the specification and goal files produced by the command is the component's name or the component's name suffixed with its tree address to make it unique, and otherwise the basename is the component's tree address. The suffix of the specification file is `.spec.sml` and that of the goal file is `.goal.sml`. Every successful translation overwrites the previous values of both of these files, so one should rename or copy the `.goal.sml` file before editing it to change it into a complete proof.

The use of the `spec` command is currently limited because the Romulus graphical interface does not have the capability to enter complete component specifications. IPSL specifications should be completed by saving the top level component with the `save` command, exiting the graphical interface, and then editing the `.ipsl` files with a text editor. If you then restart the graphical interface using the saved component, the completed IPSL specifications will be available to the `spec` command. Attempts to use the `spec` command without first completing all the IPSL specifications of the components will result in an error. Note that *all* the information contained in an `.ipsl` file is stored internally in the graphical interface, but the graphical interface can only be used to display or modify some of this information.

This limitation can be avoided, though, by using the `ipsl2hol` translator supplied with Romulus. This translator performs the same function as the `spec` command, but operates directly on IPSL specification files rather than on the graphical interface's component data structures. It parses the IPSL specification file, produces a corresponding component data structure, and produces the appropriate specification and goal files for this component. By using `ipsl2hol` you can avoid reentering the graphical interface to translate

the IPSL specifications. If the IPSL file is a specification of a composite process, each of its subcomponents is also translated. We recommend that `ips12hol` always be invoked on the top-level process; this insures that global files are always created and referenced correctly in the HOL translations.

In addition to the possible failures for the `spec` command, the `ips12hol` translator fails if it cannot open the file containing the IPSL specification or if this file contains an IPSL syntax error.

The one exception to the preference of using the `ips12hol` command over the `spec` button is when you define your own levels. (See Appendix C for an example.) The `ips12hol` command cannot be used when the specifications contain user defined levels; the `spec` button must be used instead. As with the `ips12hol` command, we recommend that the `spec` button always be invoked on the top-level process.

### 3.5.9 check

This command checks whether the `rtheory` file associated with the component, selected by clicking the left mouse button, is consistent with the Romulus graphic's description of that component, and if so whether this `rtheory` file identifies this component as proved secure. The `rtheory` file the command associates with a component is the file whose basename is the component's name and whose extension is `.rth`; the command causes an appropriate error message to appear and has no further effect if the selected component is unnamed.

The command checks, for the theory summarized in the component's `rtheory` file and its ancestors, that the following consistency conditions hold:

- the set of possible security levels equals the set of security levels previously supplied to Romulus through its built-in defaults or the `rtheory` file whose basename is the `levelfile` parameter;
- the dominance relation is consistent with the dominance relation previously supplied to the graphical interface in the same way;
- input ports are the same as the input ports on the component;
- output ports are the same as the output ports on the component;



- the range of security levels for messages through each port includes the range of security levels Romulus assigns to that port; and
- the theory identifies as distinct every pair of distinct ports on the component.

If all these conditions are satisfied and the rtheory file identifies the component as proved secure, the command sets the component's status accordingly and redraws the display to confirm its action. A process is, in this context, considered to have been proven secure if the appropriate conditions for the process have been proved.

If the selected component does not have an associated rtheory file, if it has a port with no name or no explicit security-level limits set, if it has two ports with the same name (a situation not modeled in rtheory files), if some error occurs in extracting information from the rtheory file or one of its ancestors, if the rtheory file does not identify the component as proved secure, or if one of the six consistency conditions given above is not satisfied, the command causes an appropriate error widget to appear but has no other effect.

### 3.5.10 display

This command allows the user to display information about a component. The information includes the component's name if it has one, and its tree address otherwise; the name or tree address of its parent; the names or tree addresses of its children, if any; and the names or tree addresses of its ports, if any.

The user selects the component to be displayed, either the open component or one of its children, by clicking the left mouse button. The command then causes a window to appear, overwriting a portion of the canvas window. Pressing any mouse button with the mouse cursor on the canvas then removes the information window and has no further effect.

The display window contains a scrollbar in the rare cases when all of the information will not fit on a single page. On the scrollbar, clicking the left mouse button views a later part of the text, clicking the right mouse button views an earlier part of the text, and clicking the middle mouse button views a part of the text whose position is proportional to the mouse cursor's position on the scrollbar.

## 3.6 Port Commands

This section describes the third row of the command buttons, the port commands. These modify or examine the ports owned by the currently open process or its children and modify the connections between these ports.

### 3.6.1 create/connect

This command allows the user to create a port without connecting it to any other ports, to connect two existing ports, to create a port and connect it to an existing port, or to create and connect two new ports. Pressing and releasing the left or middle mouse button, possibly after holding this button and moving the mouse, determines a pair of locations for new or existing ports. If these locations are sufficiently close together, as when the button is clicked and the mouse is not moved, the two are taken to be a single location and a port is created at that location unless one already exists there. If the two locations select two existing ports, then these ports are connected. If the locations select one existing port and give one new port location, then the new port is created and connected to the existing port. If the locations give two new port locations, then these ports are created and connected.

If at least one existing port is selected, the type of any new port created and the direction of data flow between this new port and the existing one are determined by the type of the existing port and the parent-child or sibling relationships between the ports's owners. In this case, the left and middle mouse buttons are equivalent.

Otherwise, clicking the left mouse button creates an output port, and creating a pair of ports with the left mouse button creates ports of whatever types give a data flow from the button-press port to the button-release port. Clicking the middle mouse button creates an input port, and creating a pair of ports with the middle mouse button creates ports of whatever types give a data flow to the button-press port from the button-release port. In these other cases, the types of the ports created will depend on the choice of mouse button and/or the parent-child or sibling relationships between the port's owners.

No ports are actually created until the left or middle mouse button is released, and then ports are created only if the indicated connection between ports can be made without error. Several types of errors are possible. Ports

are also typically not created at the exact locations where mouse-button presses or releases occur, but at nearby locations determined by projecting the button-press locations to the nearest valid port locations. The command shows a port icon of indeterminate type at the location where a port will be created.

Pressing or releasing the left or middle mouse button with the cursor on a canvas location determines the location of an existing port or a desired new port as follows: If the location is inside a child component of the open component, it is projected onto the nearest edge of that child component. Otherwise, if the location is outside or near the edge of the open component—“near” is currently defined as within eight times the distance from the open component’s edge to the edge of the canvas—it is projected inward or outward to the nearest edge of the open component. If the projected locations determined by the press and release of a single mouse button are within one port-icon width of each other, the two events are taken to determine only a single port location.

If a projected location is inside the rectangle bounding the icon for an existing port, the selection of that location is taken as the selection of that existing port. Otherwise, the projected location is the center of the icon for a new port to be created. If the projected location is on the edge of the open component, the new port is owned by the open component; if it is on the edge of one of the open component’s children, the new port is owned by that child.

Each time the command creates a new port, it resets that port’s owner’s status, and the statuses of all that owner’s ancestors, to “possibly insecure”. If it connects ports owned by two sibling components, it resets the statuses of their parent and its ancestors to “possibly insecure”. If it connects ports owned by parent and child components, it resets the statuses of the parent and its ancestors to “possibly insecure”.

The command causes an error window to appear and has no further effect if the location of a port to be created is such that it would overlap with an existing port, if the location is inside the open component but not in a child component or close to an edge of the open component, if an attempt is made to reconnect to a port that is already connected, if an attempt is made to connect two ports owned by the same component, or if an attempt is made to connect two ports whose types and owners are such that they cannot be validly connected.

### **3.6.2 move**

This command allows the user to move ports to other points on the boundary of the same owner. The user selects the port to be moved by clicking the left mouse button. Clicking the middle mouse button determines the location to which the port is moved, a location determined by projecting onto the nearest edge of an enclosing component, if any, as described above for the `create/connect` command.

### **3.6.3 delete**

This command allows the user to delete ports and/or connections between them. With the left mouse button, selecting a port deletes the port, and selecting a connection deletes the connection. With the middle mouse button, selecting a connection deletes the connection and the ports at both its ends. Deleting a port automatically deletes any connections to the port.

If the command deletes a port, it resets the statuses of the port's owner and that owner's ancestors to "possibly insecure". If it deletes a connection between two ports owned by sibling components, it resets the statuses of their owner and all of its ancestors to "possibly insecure". If it deletes a connection between ports owned by parent and child components, it resets the statuses of the parent and all its ancestors to "possibly insecure". If it deletes a port or connection involved in a displayed possibly insecure data flow, it clears the display of this flow.

### **3.6.4 display**

This command allows the user to display information about a port. The information includes the port's name if it has one, and its tree address otherwise; its type (input or output); the name or tree address of the port's owner; any lower bound on the security levels of users or processes capable of accessing messages passing through the port; any upper bound on the security levels of users or processes capable of creating messages passing through the port; and the names or tree addresses of the ports it is connected to, if any.

The user selects the port to be displayed by clicking with the left mouse button. The command then causes a window to appear, overwriting a portion

of the canvas window. Pressing any mouse button with the mouse cursor on the canvas then removes the information window and has no further effect.

### 3.6.5 modify

This command allows the user to modify, for one or two ports at a time, the upper and lower bounds on the security levels of messages passing through the port. The command is most useful for setting security-level limits that are the same for several ports.

The command causes a text-entry window to appear in which the user enters the desired text string and indicates its use by clicking on one of the subcommand buttons. (The default strings are `unclassified`, `confidential`, `secret`, and `top_secret`.) Afterwards, the user selects a port by clicking with the left mouse button or selects a connection between ports by clicking with the middle mouse button. If the user selects a port, and no errors arise for changing that port, that port is modified. If the user selects a connection, and no errors arise for changing either of the ports at the ends of that connection, both of these ports are modified.

The command causes an appropriate error message to appear and has no further effect if the user attempts to modify a port before entering a text string and indicating its use, if the user attempts to use a string as a security limit and this string was not previously specified as a possible security level, if the user attempts to specify two security limits such that the lower bound is not less than or equal to the upper bound, or if the user attempts to change level ranges so that the range for source data is not contained in the range for destination data.

The check that the ranges assigned to each end of a connection are consistent, that is, the check that the range assigned to the port at the tail of arrow is a subrange of the range assigned to the port at the head of the arrow can make it difficult to assign and change level ranges. This check may effect the order that Romulus allows you to change the limits on either end of a connection. For example, you may need to narrow the range at the tail end of the arrow before you narrow it at the head end. Or, you may need to widen the range at the head end of the arrow before you can widen the range at tail end of the arrow. Also, in a multi-layer model these restrictions may propagate up and down through the different layers of the model. For example, you should start at the highest affected component and work your

way down to narrow level ranges, and start at the lowest affected component and work your way up to higher level components to widen level ranges.

A left-button, single-port modification is taken as having precedence over a middle-button, double-port modification. A single-port modification can change an explicit earlier security limit value, but a double-port modification attempt causes an appropriate error message to appear and has no further effect in this situation.

When the command changes a port's lower or upper security limits, it resets that port's owner's and all that owner's ancestors' statuses to "possibly insecure". If it modifies a security-level limit for a port involved in a displayed possibly insecure data flow, it clears the display of this flow.

# Chapter 4

## Introduction to the HOL90 Environment

The Romulus tool set provides the means for constructing formal proofs of security properties. The HOL90 proof assistant system is used in Romulus for this purpose. HOL90 is a reimplementation, in Standard ML of New Jersey (SML), of the Cambridge HOL system (HOL88). This chapter provides a very brief introduction to Standard ML, the HOL Logic, and HOL's approach to formal proofs. It is intended to provide only the minimum information necessary to interact with the HOL90 system; it is not intended to be complete or comprehensive. For further information on Standard ML and HOL, refer to [8, 12, 11, 10, 9].

Romulus tools that use the HOL system are described in chapters 5 and 6. If you are familiar with HOL, you can skip this chapter without loss of continuity.

### 4.1 Introduction to Standard ML

The meta-language SML is an interactive, functional programming language that allows you to make declarations and evaluate expressions. The evaluation of an expression produces the value of the expression and its type. A declaration binds a value to a name. This section describes some of the basic data types available in SML and some of the basic things you can do in SML. For further information, see [8].

The command `hol` starts the SML interpreter and loads in all the declarations specific to HOL90.

```
% hol
```

```

HHH          LL
HHH          LL
HHH          LL
HHH          LL
HHH      0000  LL
HHHHHHH  00 00  LL
HHHHHHH  00 00  LLL
HHH      0000  LLLL
HHH          LL LL
HHH          LL LL
HHH          LL LL
HHH          LL LL90.5

```

```

Created on Thu Mar 18 14:08:34 EST 1993
using: Standard ML of New Jersey, Version 0.93, February 15, 1993

```

```

val it = () : unit
-

```

The hyphen (-) is the SML prompt, which indicates that the interpreter is ready for input. User input is in italic type. For example, if we input the constant expression

```
- 17;
```

the interpreter responds with `val it = 17 : int`. This response tells us that the interpreter has bound the value of the expression (17) to the variable `it` and that the type of this variable is `int`. The interpreter always binds the value of the last expression to the variable `it` and always tells us what the type of that expression is. If we evaluate the variable `it`, we see that it has the value 17 and is of type `int`.



```
- it;  
val it = 17 : int
```

You can bind the value of an expression to a variable of your choosing using a `val` declaration. For example,

```
- val x = 17;  
val x = 17 : int
```

SML has several other atomic types, including `real`, `bool`, and `string`, and the usual arithmetic and relational operators on them. For example,

```
- 1+3*4=17-4;  
val it = true : bool
```

SML also allows you to construct compound data types such as lists, cartesian products, and records. For example, here is a list:

```
- [1,2,3,4];  
val it = [1,2,3,4] : int list
```

The type of this list is `list of integers`. We can just as easily have a list of strings:

```
- ["cat","dog","pig"];  
val it = ["cat","dog","pig"] : string list
```

Note that all items in a list must have the same type. The functions `hd` and `tl` can be used to get the head and tail of a list.

```
- hd ["cat","dog","pig"];  
val it = "cat" : string  
- tl ["cat","dog","pig"];  
val it = ["dog","pig"] : string list
```

The cons constructor (`::`) can be used to add an item to the beginning of a list:

```
- 1::[2,3];  
val it = [1,2,3] : int list
```

SML also allows the construction of cartesian products, or n-tuples. For example,

```
- (1,2,3);  
val it = (1,2,3) : int * int * int
```

creates a 3-tuple consisting of three integers. The items in a tuple do not need to be of the same type

```
- ("bob",17,true);  
val it = ("bob",17,true) : string * int * bool
```

is a 3-tuple consisting of a string, an integer, and a boolean. Individual items in a tuple can be extracted by binding the tuple to a pattern of the same form. For example,

```
- val (name,age,flag) = ("bob",17,true);  
val name = "bob" : string  
val age = 17 : int  
val flag = true : bool
```

binds the string "bob" to the variable `name`, the integer 17 to the variable `age`, etc. The arity of the pattern and the tuple must be the same.

SML also has a record type that is similar to a tuple type except that the individual fields are named.

```
- val emprec = {name="bob",age=17,flag=true};  
val emprec = {age=17,flag=true,name="bob"} :  
              {age:int, flag:bool, name:string}
```

The order in which the fields are specified does not matter, that is, `{name="bob",age=17,flag=true}` is the same as `{age=17,flag=true,name="bob"}`. Individual fields can be extracted from this record by applying the operators `#age`, `#flag`, and `#name` to the record. For example,

```
- #name emprec;
val it = "bob" : string
```

Tuples are implemented in SML as records whose fields are named by the integers, as in

```
- #1 ("bob", 17, true);
val it = "bob" : string
```

SML functions are first-class objects, meaning that they can be created and manipulated like other data objects. For example, you can use lambda notation to create a function that adds two integers as follows:

```
- fn (a:int,b:int) => a+b;
val it = fn : int * int -> int
```

(The keyword `fn` takes the place of `lambda` in lambda notation.) This function can be bound to a variable and applied to a 2-tuple as follows:

```
- val add = fn (a:int,b:int) => a+b;
val add = fn : int * int -> int
- add(1,2);
val it = 3 : int
```

A more convenient means of defining this function uses the following shorthand notation:

```
- fun add(a:int,b:int) = a+b;
val add = fn : int * int -> int
```

which defines the same function. The type declarations on the function parameters are necessary, in this case, in order for the SML interpreter to determine the types of the parameters. The reason is that the addition operator can be applied equally well to ints or to reals and the interpreter does not have enough information to choose between these two types.

Where possible, SML will automatically infer types, so that in many cases variables need not be explicitly typed. For example, the concatenation operator (^) applies only to strings, so the following definition

```
- fun join a b = a^b;  
val join = fn : string -> string -> string
```

does not require type declarations.

The add function defined above takes a 2-tuple as its argument. It is more typical to define *curried* functions. For example,

```
- fun add (a:int) (b:int) = a+b;  
val add = fn : int -> int -> int  
- add 1 2;  
val it = 3 : int
```

It would appear that add is a function that takes two integers as arguments and returns an integer. In fact, add is a function that takes an integer as its argument and returns a function. This new function takes an integer as its argument and returns a integer. We can bind the function returned by add to its own variable, creating a new function that we can then apply. For example, the following

```
- val inc = add 1;  
val inc = fn : int -> int  
- inc 17;  
val it = 18 : int
```

creates a new function inc that increments its argument by 1.

Local declarations can be introduced in an expression. For example, in the expression

```
- let val x = [1,3,4] in hd x :: 2 :: tl x end;
val it = [1,2,3,4] : int list
```

`val x = [1,3,4]` is a local declaration whose scope extends from the `in` to the `end`.

## 4.2 The HOL Logic

The HOL90 system supports higher order logic, a version of predicate calculus that is typed and allows variables to range over functions and predicates. The HOL logic is summarized in the Table 4.1.

Kind of term	HOL notation	Kind of term	HOL notation
Truth	<code>T</code>	Equality	<code>t1 = t2</code>
Falsity	<code>F</code>	$\forall$ -quantification	<code>!x.t</code>
Negation	<code>~t</code>	$\exists$ -quantification	<code>?x.t</code>
Disjunction	<code>t1 \\/ t2</code>	$\epsilon$ -term	<code>@x.t</code>
Conjunction	<code>t1 /\ t2</code>	Conditional	<code>(t ==&gt; t1   t2)</code>
Implication	<code>t1 ==&gt; t2</code>		

Table 4.1: The HOL Logic

### 4.2.1 HOL Terms

Logic expressions in HOL90 have the SML abstract type `term`. Logic expressions can be expressed using the HOL object language, which is in turn parsed by a parser to produce the corresponding term. HOL object language expressions are written inside of a pair of single quotation marks and must be explicitly parsed using the HOL90 function `term_parser`. For example, the conjunction of `a` and `b` is represented by the following SML expression:

```
- --'a /\ b'--;
val it = (--'a /\ b'--) : term
```

The object language expression 'a /\ b' represents the conjunction of a and b. The two pairs of hyphens (--) invoke `term_parser` to parse the expression.

Each HOL term also has a HOL type. HOL types are also written inside single quotes and are parsed using the HOL90 function `type_parser`. For example, in

```
- == ':bool' ==;
val it = (==':bool'==) : hol_type
```

' :bool ' is HOL object language for a boolean type and the two pairs of equal signs (==) invoke the function `type_parser`.

The HOL type of a HOL term can be determined using the function `type_of`.

```
- type_of (--'a /\ b'--);
val it = (==':bool'==) : hol_type
```

A HOL term can be a constant, a variable, a  $\lambda$ -expression, or a function application. Here are some examples of constant terms:

```
- --'17'--;
val it = (--'17'--) : term
- type_of it;
val it = (==':num'==) : hol_type
- --'T'--;
val it = (--'T'--) : term
- type_of it;
val it = (==':bool'==) : hol_type
- --'F'--;
val it = (--'F'--) : term
- type_of it;
val it = (==':bool'==) : hol_type
```

The type of a variable may be given explicitly:

```

- --'x:bool'--;
val it = (--'x'--) : term
- type_of it;
val it = (=='bool'==) : hol_type

```

Or it may be inferred by the parser:

```

- --'~x'--;
val it = (--'~x'--) : term
- type_of it;
val it = (=='bool'==) : hol_type

```

If there is insufficient information for the parser to infer the type of a variable, an error results.

```

- --'x'--;
Unconstrained type variable in the variable
  (x :?1)

uncaught exception HOL_ERR

```

The following  $\lambda$ -expression defines a function of one variable that adds 1 to its argument.

```

- --'\x.x+1'--;
val it = (--'\x. x + 1'--) : term
- type_of it;
val it = (=='num -> num'==) : hol_type

```

The type of this function is inferred to be `num -> num` since `num` (the set of non-negative integers) is the type on which addition is performed.

The following term is a function application:

```

- --'(\x.x+1) y'--;
val it = (--'(\x. x + 1) y'--) : term
- type_of it;
val it = (=='num'==) : hol_type

```

The type of this term is the type of the result of the function, in this case `num`.

The values of SML variables can be used in object language expression using the antiquotation operator (`^`).

```
- val x = --'a /\ b'--;
val x = (--'a /\ b'--) : term
- val y = --'^x /\ c'--;
val y = (--'a /\ b /\ c'--) : term
```

The value of the SML variable `x` is included in the value of `y` by antiquoting it with the caret (`^`) character. SML variables denoting HOL types can be antiquoted as well, but their values must be wrapped in the constructor `ty_antiq`.

```
- val h = ty_antiq(==':num list'==);
val h = (--'(ty_antiq(==':num list'==)))'--' : term
- --'k:^h'--;
val it = (--'k'--') : term
- type_of it;
val it = (==':num list'==) : hol_type
```

## 4.2.2 HOL Theories

The HOL90 system is used to create *theory* objects. A HOL theory contains sets of types, constants, definitions, axioms, and theorems that have been proven from the axioms and definitions. HOL theories can be saved in theory files. For example, in HOL90 the theory `mytheory` is saved in two files `mytheory.holsig` and `mytheory.thms`. The former gives the constants defined in the theory and the latter gives the facts assumed, defined, or proved, about these constants. Theory files also contain pointers to other theory files, known as *parents* of the theory. Anything in a parent theory can be referred to in the descendant theory. This structure provides a hierarchical representation of theories.

The HOL system has two modes of operation: *draft* mode and *proof* mode. Draft mode is used to add constants, definitions, and axioms to a



theory. Proof mode is used to add theorems to a theory that can be formally proved from the definitions, axioms, and previously proved theorems of the theory. Proof mode cannot be used to add constants, definitions, or axioms to a theory.

The HOL system provides a number of functions for manipulating theories; brief descriptions of the most important of these functions follow. HOL initially starts in proof mode and the current theory is initially HOL, which contains basic facts about arithmetic, logic, lists, pairs, etc. The function `current_theory` returns a string containing the name of the current theory.

```
- current_theory();  
val it = "HOL" : string
```

You can create a new theory with the function `new_theory`, which takes a string containing the name of the new theory as its argument.

```
- new_theory "mytheory";  
  
Declaring theory "mytheory".  
  
Theory "HOL" already consistent with disk, hence not exported.  
val it = () : unit  
- current_theory();  
val it = "mytheory" : string
```

The `new_theory` function creates a new theory call `mytheory`, makes the new theory the current theory, makes the old theory the parent of the new theory, and puts HOL into draft mode. Other previously stored theories can be added as parents using `new_parent`. For example, the following

```
- new_parent "string";  
val it = () : unit  
- new_parent "integer";  
val it = () : unit
```

makes the theories `string` and `integer` parents of `mytheory`. HOL must be in draft mode to use `new_parent`.

At this point, you can add new constants, definitions, and axioms to the current theory. (Functions for these purposes are discussed in the next section.) When you are finished with this process, you can save the current theory in theory files using the following command:

```
- export_theory();  
  
Theory "mytheory" exported.  
val it = () : unit
```

The theory can then be used in a later HOL session. The `export_theory()` function can be used in either draft or proof mode. Another command

```
- close_theory();  
  
Theory "mytheory" closed.  
val it = () : unit
```

changes HOL from draft mode to proof mode. It *does not* save the theory in theory files; `export_theory` must be used for that. The `close_theory` function leaves the current theory unchanged, and you can use the previously defined constants, definitions, and axioms to prove theorems in proof mode. However, you must be sure to use `export_theory` to save any new theorems in the theory files before exiting HOL.

A theory can be loaded from its theory files using `load_theory`:

```
- load_theory "mytheory";  
  
Loading theory "mytheory"  
  
Theory "HOL" already consistent with disk, hence not exported.  
val it = () : unit
```

This function makes `mytheory` the current theory and puts HOL in proof mode. On the other hand,

```
- extend_theory "mytheory";
```

Extending theory "mytheory"

```
Theory "HOL" already consistent with disk, hence not exported.  
val it = () : unit
```

makes mytheory the current theory and leaves HOL in draft mode.

You can view the contents of mytheory with the command

```
print_theory "mytheory";
```

As a convenience, the string "-" is always taken as the name of the current theory. Thus, the current theory can be printed using the command

```
print_theory "-";
```

You can retrieve a definition or a theorem from a theory with the commands theorem and definition.

```
- theorem "arithmetic" "LESS_SUC_NOT";  
val it = |- !m n. m < n ==> ~(n < SUC m) : thm  
- definition "arithmetic" "GREATER";  
val it = |- !m n. m > n = n < m : thm
```

The first example retrieves the theorem LESS\_SUC\_NOT from the theory arithmetic and the second example retrieves the definition GREATER from the theory arithmetic.

Finally, there is an alternative way to open a new theory. HOL libraries are groups of files that have three components: logical theories, code, and help. Libraries are loaded with the command

```
load_library{lib = library; theory = "mytheory"};
```

This command puts HOL into draft mode (if necessary), opens a new theory mytheory, makes any theories in library the parents of this new theory, and performs other tasks that make the library accessible.

### 4.2.3 Defining HOL Types and Constants

HOL users can define their own concrete recursive data types using the `define_type` function. Concrete recursive types are types whose values are generated by a set of *constructors* (i.e., functions) that yield concrete representations for these values. Examples include types that denote finite sets of atomic values (enumerated types), types that denote sets of structured values (record types) or finite disjoint unions of structured values (variant records), and types that denote sets of recursive data structures such as trees (recursive types).

For example, if you are in draft mode, the definition

```
val BinTree_Def =
  define_type
    {name = "bintree_DEF",
     type_spec = 'bintree = LEAF of num |
                  NODE of bintree#num#bintree',
     fixities = [Prefix,Prefix]};
```

defines a data type of binary trees with non-negative integers at each leaf and node. The `type_spec` field defines a type `bintree` with two constructors `LEAF` and `NODE`. The `LEAF` constructor takes a single argument of type `num` and returns a binary tree consisting of a single node. For example,

```
- --'(LEAF 17)'--;
val it = (--'LEAF 17'--): term
- type_of it;
val it = (=='bintree'==): hol_type
```

The `NODE` constructor takes two binary trees and a non-negative integer as its arguments and returns a binary tree.

```
- --'(NODE (LEAF 3) 5 (LEAF 7))'--;
val it = (--'NODE (LEAF 3) 5 (LEAF 7)'--): term
- type_of it;
val it = (=='bintree'==): hol_type
```

The `fixities` field indicates that `LEAF` and `NODE` are prefix operators. The `define_type` function constructs and proves a theorem that is an abstract characterization of the data type described in this specification. This theorem is stored with the name `bintree_Def` in the current theory. In addition, `define_type` returns this theorem, which in this example is saved in the SML variable `BinTree_Def`, for ease in later use.

You can define primitive recursive functions on concrete recursive types using the function `new_recursive_definition`. For example, if you are in draft mode, the definition

```
new_recursive_definition
  {name = "Sum",
   fixity = Prefix,
   rec_axiom = BinTree_Def,
   def= --'(Sum (LEAF n) = n) /\
        (Sum (NODE t1 n t2) =
         n + (Sum t1) + (Sum t2))'--};
```

defines a function `Sum` that adds up all the numbers in a binary tree. The `name` field names the new definition in the current theory where the results of the definition will be stored. The `fixity` field determines that `Sum` will be a prefix operator. The `rec_axiom` field is the name of the theorem describing the concrete recursive type for binary trees. The `def` field gives the actual definition of the `Sum`. The `new_recursive_definition` function returns a theorem that states the requested definition. With this definition `Sum` can be used just like any other function.

```
- --'(Sum (LEAF 5))'--;
val it = (--'Sum (LEAF 5)'--): term
- type_of it;
val it = (==':num'==): hol_type
- --'(Sum (NODE (LEAF 1) 2 (LEAF 3)))'--;
val it = (--'Sum (NODE (LEAF 1) 2 (LEAF 3))'--): term
- type_of it;
val it = (==':num'==): hol_type
```

You can declare new constants using `new_constant`. For example, if you are in draft mode, the function

```

- new_constant{Name="X",Ty==':num'==};
val it = () : unit
- --'X'--;
val it = (--'X'--) : term
- type_of it;
val it = (==':num'==) : hol_type

```

makes X a constant of type num in the current theory.

The above declaration says nothing about the constant other than its type. Using `new_definition` you can declare a constant together with a definitional fact about the constant. For example, if you are in draft mode, the following

```

- new_definition("x",--'x=17'--);
val it = |- x = 17 : thm
- --'x'--;
val it = (--'x'--) : term
- type_of it;
val it = (==':num'==) : hol_type
- definitions "-";
val it = [("x",|- x = 17)] : (string * thm) list

```

declares a constant `x` of type `num` and stores an assertion in the current theory that `x` has a value equal to 17.

HOL also allows using `define_type` as a polymorphic type constructor through the use of type variables. Consider the binary tree example. Instead of defining a binary tree of `nums`, we could have defined binary trees of objects of arbitrary type this way:

```

val BinTree_Def =
  define_type {
    name = "bintree_DEF",
    type_spec = 'bintree = LEAF of 'a |
                NODE of bintree#'a#bintree',
    fixities = [Prefix,Prefix]
  };

```

The variable `'a` is an example of a type variable. This definition of a binary tree allows a node or leaf to contain an arbitrary type, as long as the same

type is stored at each node and leaf of the tree. For example, the binary tree

```
- --'(LEAF 17)'--;
val it = (--'LEAF 17'--): term
- type_of it;
val it = (==':num bintree'==): hol_type
```

contains numbers; its type is num bintree. The binary tree

```
- --'(LEAF (3,14))'--;
val it = (--'LEAF (3,14)'--): term
- type_of it;
val it = (==':(num # num) bintree'==): hol_type
```

contains pairs of numbers; its type is (num # num) bintree. The binary tree

```
- --'(NODE (LEAF (1,2,3)) (2,1,3) (LEAF (1,3,4)))'--;
val it = (--'NODE (LEAF (1,2,3)) (2,1,3) (LEAF (1,3,4))'--): term
- type_of it;
val it = (==':(num # num # num) bintree'==): hol_type
```

contains 3-tuples of nums. Its type is (num # num # num) bintree.

It is also possible to explicitly instantiate a type. For example, the expression (num)bintree instantiates the type num bintree.

```
- val numtree = ty_antiq(==':(num)bintree'==);
val numtree = (--'(ty_antiq(==':num bintree'==))'--): term
- --'t:~numtree'--;
val it = (--'t'--): term
- type_of it;
val it = (==':num bintree'==): hol_type
```

By wrapping the ty\_antiq constructor around the type definition and saving the definition in an SML variable numtree, we can quote this definition in later expressions, as was done above to declare t to be of type num bintree.

## 4.3 Goal Oriented Proof: Tactics and Tacticals

The HOL system allows the creation of only well-formed theories, that is, all theorems in the theory must be logical consequences of the definitions and axioms of the theory. To add a theorem to a theory you must construct a formal proof of its correctness. HOL supports both forward proof and backward proof. In a forward proof, new theorems are derived from previously existing axioms, definitions, or theorems. In a backward proof, the user starts with a desired theorem and reduces it to existing axioms, definitions, or theorems. Forward proof is the underlying means for all theorem proving in HOL; backward proof, though, is usually more convenient. Backward proof gives a natural organization to the proof-search process and saves having to repeatedly enter the assertions being proved.

Here are two examples of backward proof:

- To prove  $t1 \wedge t2$ , it is sufficient to prove  $t1$  and prove  $t2$ .
- To prove  $t1 \implies t2$ , it is sufficient to prove  $t2$  from the assumption  $t1$ .

HOL90 provides support for having the user guide the creation of new theorems by backward proof. This support is in the form of a *goal stack*, *tactics*, and *tacticals*. After the user has completed a backward proof, HOL90 then automatically applies the corresponding forward inference rules to produce the theorem proved. We will describe goals, tactics, the goal stack, and tacticals, then give several examples of proofs of simple theorems.

### 4.3.1 Goals

A *goal* is an SML value isomorphic to, but distinct from, the SML abstract type `thm` of theorems. A goal consists of a list of assumptions and a conclusion. In HOL, an assumption or conclusion is just a term of type `:bool`, so in SML a goal is just a pair consisting of a term list and a term. For instance,

```
([--'a < b'--], --'0 < a'--], --'0 < b'--)
```

is a goal. A goal gives the form of a desired theorem. The theorem corresponding to the above goal is



$[a < b, 0 < a] \vdash 0 < b$

### 4.3.2 Tactics

A *tactic* is an SML function that, when applied to a goal, returns a list of subgoals and a *justification*. A justification is an SML function, typically a forward rule of inference, that when applied to the list of theorems corresponding to the list of subgoals produces the theorem corresponding to the original goal. For example, for the goal

$(\square, \text{--}'T \wedge (!n. 0 \leq n)\text{'--})$

the tactic `CONJ_TAC` returns the list of subgoals

$[(\square, \text{--}'T\text{'--}), (\square, \text{--}'!n. 0 \leq n\text{'--})]$

and a justification that uses the forward rule of inference `CONJ` to map the theorems  $\vdash T$  and  $\vdash (!n. 0 \leq n)$  to the theorem  $\vdash T \wedge (!n. 0 \leq n)$ . A tactic *solves* a goal if it reduces the goal to the empty list of subgoals.

Here are the most frequently used tactics and informal descriptions of what they do. Examples of the use of each of these tactics can be found in section 4.3.5. See the HOL documentation for formal, detailed descriptions.

- `ACCEPT_TAC` solves a goal if the goal is the same as a supplied theorem after possibly renaming variables. `MATCH_ACCEPT_TAC` is a variation on this tactic that allows substituting terms for variables in matching the goal and the supplied theorem.
- `GEN_TAC` strips off the outermost universal quantifier; for proving  $\forall x. P\ x$  it suffices to prove  $P\ x$  for an arbitrary  $x$ .
- `CONJ_TAC` splits a goal  $t1 \wedge t2$  into the two subgoals  $t1$  and  $t2$ .
- `EQ_TAC` splits a goal  $u = v$ , where  $u$  and  $v$  both have type `:bool`, into the two subgoals  $u ==> v$  and  $v ==> u$ .
- `DISCH_TAC` changes a goal  $(A, u ==> v)$  into  $(A \cup u, v)$ .
- `STRIP_TAC` removes one outer connective from the goal using `CONJ_TAC`, `DISCH_TAC`, or `GEN_TAC`.

- ASSUME\_TAC adds the conclusion of a theorem to a goal's assumptions.
- EXISTS\_TAC, for a term  $u$ , changes a goal's conclusion from  $?x. t[x]$  to  $t[u]$ .
- IMP\_RES\_TAC takes an implicative theorem of the form  $\vdash u \Rightarrow v$ , matches each of a goal's assumptions against  $u$  (possibly making appropriate variable substitutions), and if a match succeeds, adds  $v$  (possibly with appropriate substitutions for variables) to the goal's list of assumptions. IMP\_RES\_TAC deduces consequences of a known theorem and the goal's assumptions.
- RES\_TAC takes each of a goal's assumptions of the form  $\vdash u \Rightarrow v$ , matches each of a goal's remaining assumptions against  $u$  (possibly making appropriate variable substitutions), and if a match succeeds, adds  $v$  (possibly with appropriate substitutions for variables) to the goal's list of assumptions. RES\_TAC deduces consequences of pairs of the goal's assumptions.
- REWRITE\_TAC transforms, or solves, a goal by using the conclusions of a list of equational theorems as rewrite rules (i.e., as left-to-right replacement rules). It applies these rewrites recursively, and to arbitrary depth, and automatically makes appropriate substitutions for variables in the equational theorems to produce matches with terms in the goal. Further, REWRITE\_TAC automatically applies common tautologies (e.g.,  $\vdash T \wedge t = t$ ) to further simplify the goal, and it takes conclusion terms  $t$  to be the equation  $t = T$ , so it solves goals that match the conclusions of arbitrary theorems. It can be used in place of MATCH\_ACCEPT\_TAC, but is more computationally expensive.

There are a number of variations on REWRITE\_TAC. PURE\_REWRITE\_TAC uses only the list of equational theorems as rewrite rules. ONCE\_REWRITE\_TAC applies the rewrite rules one time only. PURE\_ONCE\_REWRITE\_TAC is a combination of the above two.

- ASM\_REWRITE\_TAC acts as REWRITE\_TAC does, but it also uses the goal's assumptions as rewrite rules.

There are a number of variations on ASM\_REWRITE\_TAC. PURE\_ASM\_REWRITE\_TAC, ONCE\_ASM\_REWRITE\_TAC, and PURE\_ONCE\_ASM\_REWRITE\_TAC

are similar to the corresponding variations on `REWRITE_TAC`.

- `MP_TAC` reduces a goal to an implication from a known theorem. `MATCH_MP_TAC` is a variation on this tactic that uses more general matching techniques than `MP_TAC`.

`ASM_REWRITE_TAC` and `REWRITE_TAC` are big hammers; they are very often the tactics that solve final subgoals.

### 4.3.3 Goal Stack

The *goal stack* maintains a record of all current subgoals and all current justifications that relate subgoals to earlier subgoals. The following functions are the most important ones for manipulating the goal stack:

- Function `set_goal` puts a new goal onto the top of the goal stack.
- Function `g`, applied to a conclusion, puts the goal of showing this conclusion from an empty list of assumptions onto the top of the goal stack; `g conclusion` is equivalent to `set_goal([], conclusion)`.
- Function `expand`, abbreviated as `e`, takes a tactic, applies it to the top subgoal, adds any subgoals returned by the tactic to the stack of subgoals, saves the justification returned by the tactic on the subgoal stack, and if the tactic solves the top subgoal, automatically applies the saved justifications to produce the theorem corresponding to the top subgoal.
- Function `rotate`, abbreviated as `r`, applied to an integer, rotates the top list of subgoals on the stack by that integer, so the subgoals of a common goal can be considered in any order.
- Function `top_goal`, with no arguments, returns the top goal on the goal stack.
- Function `backup`, abbreviated as `b`, with no arguments, “backs up” and undoes the effect of the last command that affected the goal stack.
- Function `save_top_thm`, with a string argument, stores the theorem just proved with the subgoal package in the current theory, using the string as the theorem’s name.

### 4.3.4 Tacticals

A *tactical* is an SML function that takes a tactic (or tactics) as arguments and returns a tactic (or tactics) as a result. Here are the most important tacticals and informal descriptions of what they do:

- THEN is an infix operator that takes two tactics, applies the first one to a goal, and then applies the second one to each of the subgoals, if any, produced by the first one. THEN is very convenient for proving many subgoals of the same form.
- REPEAT repeatedly applies a tactic until the tactic fails. (Failure occurs when a goal is not of the form expected by the tactic.) The best first step in proving a goal is often to use REPEAT STRIP\_TAC.
- ORELSE is an infix operator that takes two tactics, applies the first to a goal unless that fails, and if that fails, applies the second to the goal.

### 4.3.5 Examples

Here, then, are several examples of using the goal stack, tactics, and tacticals in HOL90. The lines entered by the user are those that begin with a hyphen (-) and are in italic type.

The first example demonstrates the use of ACCEPT\_TAC. Our goal is to prove that all boolean variables  $x$  either are true or are false. Fortunately, a theorem to this effect has already been proved and saved in the SML variable BOOL\_CASES\_AX. The only difference between this theorem and our goal is the use of a different variable. We start by entering our goal; the interpreter responds by printing the goal in a standard form. The conclusion is printed, followed by a row of equal signs (=), then followed by the assumptions. In this case there are no assumptions so a blank line is printed.

```
- g '!x. (x=T) \\/ (x=F)';  
(-- '!x. (x = T) \\/ (x = F) '---)  
=====
```

```
val it = () : unit
```

Next we print the theorem `BOOL_CASES_AX`. This is done for informational purposes only; this action has no effect on the proof or the goal stack and could just as well have been left out. After this, the tactic is applied.

```
-  BOOL_CASES_AX;
val it = |- !t. (t = T) \\/ (t = F) : thm
-  e(ACCEPT_TAC BOOL_CASES_AX);
OK..

Goal proved.
|- !t. (t = T) \\/ (t = F)

Top goal proved.
val it = () : unit
```

The tactic `ACCEPT_TAC` performs the necessary variable substitution and uses the theorem to prove the goal. Note in this last example that the goal stack is affected only by the functions that explicitly manipulate it, so evaluating the SML variable `BOOL_CASES_AX` has no effect on the goal stack.

The following goal contains the universal quantifier (!).

```
-  g '!x. 0 <= x';
(---'!x. 0 <= x'---)
=====

val it = () : unit
-  e GEN_TAC;
OK..
1 subgoal:
(---'0 <= x'---)
=====

val it = () : unit
```

The tactic `GEN_TAC` strips off the outermost universal quantifier from a goal. To prove a statement for all  $x$  it is sufficient to prove it for an arbitrary  $x$ .

The next goal is a conjunction.

```

- g 'A ∧ B';
(--'A ∧ B'--)
=====

val it = () : unit
- e CONJ_TAC;
OK..
2 subgoals:
(--'B'--)
=====

(--'A'--)
=====

val it = () : unit

```

The tactic `CONJ_TAC` makes this goal into two subgoals. If both these subgoals can be proved, then so can the original goal.

The next goal claims the equality of two terms.

```

- g '(a ∧ b) = c';
(--'a ∧ b = c'--)
=====

val it = () : unit
- e EQ_TAC;
OK..
2 subgoals:
(--'c ==> a ∧ b'--)
=====

(--'a ∧ b ==> c'--)
=====

val it = () : unit

```

EQ\_TAC replaces this equality with two subgoals, each of which is an implication. We next apply DISCH\_TAC to the topmost goal.

```
- e DISCH_TAC;
OK..
1 subgoal:
(--'c'--)
=====
  (--'a /\ b'--)

val it = () : unit
```

DISCH\_TAC adds the left-hand side of the implication to the (initially empty) set of assumptions, leaving only the right side of the implication as the new goal.

Next we have a more complicated goal.

```
- g '!x y z. x ==> (y ==> (y /\ z))';
(--'!x y z. x ==> y ==> y /\ z'--)
=====

val it = () : unit
- e(REPEAT STRIP_TAC);
OK..
2 subgoals:
(--'z'--)
=====
  (--'x'--)
  (--'y'--)

(--'y'--)
=====
  (--'x'--)
  (--'y'--)

val it = () : unit
```

Repeated application of the tactic `STRIP_TAC` is used to simplify this goal. In this case, the same results could have been achieved by three applications of `GEN_TAC`, two applications of `DISCH_TAC` and finally one application of `CONJ_TAC`.

The next goal states that there exists an  $n$  that satisfies a certain condition.

```
- g '?n. (0 < n) /\ (n <= SUC 0)';
  (--'?n. 0 < n /\ n <= SUC 0'--)
```

```
=====

val it = () : unit
- e(EXISTS_TAC (--'1'--));
OK..
1 subgoal:
  (--'0 < 1 /\ 1 <= SUC 0'--)
```

```
=====

val it = () : unit
```

Here `EXISTS_TAC` is used to substitute a particular value (1) for the variable  $n$ . The quantifier is no longer needed.

The next goal states that a boolean variable cannot be equal to its logical negation.

```
- g '~(t = ~t)';
  (--'~(t = ~t)'--)
```

```
=====

val it = () : unit
```



```

-   BOOL_CASES_AX;
val it = |- !t. (t = T) \\/ (t = F) : thm
-   e(ASSUME_TAC BOOL_CASES_AX);
OK..
1 subgoal:
(--'~(t = ~t)'--
=====
  (--'!t. (t = T) \\/ (t = F)'--

val it = () : unit

```

The theorem `BOOL_CASES_AX` might be useful in proving this goal. The tactic `ASSUME_TAC` adds this theorem to the assumptions of the goal.

The next goal has a non-empty list of assumptions. The theorem `LESS_SUC_NOT` is an implication whose left-hand side matches the assumption.

```

-   set_goal([--'m < n'--], --'n <= SUC m'--);
(--'n <= SUC m'--
=====
  (--'m < n'--

val it = () : unit
-   theorem "arithmetic" "LESS_SUC_NOT";
val it = |- !m n. m < n ==> ~(n < SUC m) : thm
-   e(IMP_RES_TAC (theorem "arithmetic" "LESS_SUC_NOT"));
OK..
1 subgoal:
(--'n <= SUC m'--
=====
  (--'m < n'--
  (--'~(n < SUC m)'--

val it = () : unit

```

`IMP_RES_TAC` adds the conclusion of this theorem to the assumptions of the goal.

Other tactics can also be used to enrich a list of assumptions. Consider the following goal.

```

- set_goal([--'a==>(b==>c)'--,--'a:bool'--,--'b:bool'--,--'a /\ c'--]);
(--'a /\ c'--)
=====
  (--'b'--)
  (--'a'--)
  (--'a ==> b ==> c'--)

val it = () : unit
- e RES_TAC;
OK..
1 subgoal:
(--'a /\ c'--)
=====
  (--'b'--)
  (--'a'--)
  (--'a ==> b ==> c'--)
  (--'c'--)

val it = () : unit

```

RES\_TAC adds the term *c* to the list of assumptions, since it can be derived from the other assumptions. We can finish this example using ASM\_REWRITE\_TAC.

```

- e(ASM_REWRITE_TAC []);
OK..

Goal proved.
.. |- a /\ c

Goal proved.
... |- a /\ c

Top goal proved.
val it = () : unit

```

The multiple “goal proved” messages show several theorems, including the one corresponding to the top goal, being proved from the justifications accumulated in the goal stack.

This example shows the power of REWRITE\_TAC.

```

- g '!m n. (m > n) ==> (m >= n)';
(--'!m n. m > n ==> m >= n'--)
=====

val it = () : unit
- val GREATER_OR_EQ = definition "arithmetic" "GREATER_OR_EQ";
val GREATER_OR_EQ = |- !m n. m >= n = m > n \ / (m = n) : thm
- val GREATER = definition "arithmetic" "GREATER";
val GREATER = |- !m n. m > n = n < m : thm
- OR_INTRO_THM1;
val it = |- !t1 t2. t1 ==> t1 \ / t2 : thm
- e(REWRITE_TAC [GREATER, GREATER_OR_EQ, OR_INTRO_THM1]);
OK..

Goal proved.
|- !m n. m > n ==> m >= n

Top goal proved.
val it = () : unit

```

Note in this last example that some basic theorems do not belong to any theory and that HOL90 has already bound SML identifiers to these theorems. Although they are not used in this example, HOL90 also has functions for binding SML identifiers to all the theorems and/or all the definitions in arbitrary theories.

The following goal states that 0 is less than the successor of any non-negative number.

```

- g '0 < m + 1';
(--'0 < m + 1'--)
=====

val it = () : unit

```

```

- val ADD1 = (theorem "arithmetic" "ADD1");
val ADD1 = |- !m. SUC m = m + 1 : thm
- e(MP_TAC ADD1);
OK..
1 subgoal:
(--'(!m. SUC m = m + 1) ==> 0 < m + 1'--)
=====

val it = () : unit

```

The tactic MP\_TAC makes this goal into an implication by making the conclusion of the theorem ADD1 the antecedent of the implication and the original goal the conclusion of the implication.

We conclude this section with a longer example. We start with our initial goal and apply CONJ\_TAC to it.

```

- g '(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])';
(--'(HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])'--)
=====

val it = () : unit
- e CONJ_TAC;
OK..
2 subgoals:
(--'TL [1; 2; 3] = [2; 3]'--)
=====

(--'HD [1; 2; 3] = 1'--)
=====

val it = () : unit

```

We can undo the last operation using backup.

```

- backup();
(--'(HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])'--)
=====

val it = () : unit

```

Reapplying CONJ\_TAC we get.

```

- e CONJ_TAC;
OK..
2 subgoals:
(--'TL [1; 2; 3] = [2; 3]'--)
=====

(--'HD [1; 2; 3] = 1'--)
=====

val it = () : unit

```

We can change the order of the subgoals using rotate.

```

- rotate 1;
(--'HD [1; 2; 3] = 1'--)
=====

(--'TL [1; 2; 3] = [2; 3]'--)
=====

val it = () : unit

```

Finally we use REWRITE\_TAC to apply the definitions of TL and HD.

```

- e(REWRITE_TAC [definition "list" "TL"]);
OK..

Goal proved.
|- TL [1; 2; 3] = [2; 3]

Remaining subgoals:
(--'HD [1; 2; 3] = 1'--)
=====

val it = () : unit
- e(REWRITE_TAC [definition "list" "HD"]);
OK..

Goal proved.
|- HD [1; 2; 3] = 1

Goal proved.
|- (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])

Top goal proved.
val it = () : unit

```

Note in this last example that for HOL object-language lists, the “head” operation, the “tail” operation, and the list separator are HD, TL, and the semicolon; for SML lists, the “head” operation, the “tail” operation, and the list separator are hd, tl, and the comma. Also note that the proof given was much longer than necessary, being used to illustrate the b and r commands. The same result could have been proved with the application of a single tactic.

```

- g '(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])';
(--'(HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])'--)
=====

val it = () : unit

```

```

- e(REWRITE_TAC[definition "list" "HD", definition "list" "TL"]);
OK..

Goal proved.
|- (HD [1; 2; 3] = 1) /\ (TL [1; 2; 3] = [2; 3])

Top goal proved.
val it = () : unit

```

### 4.3.6 More HOL Help

This chapter provides only the minimum information necessary to interact with the HOL90 system; it is not intended to be complete or comprehensive. Many resources are available to the reader who wishes to know more. Additional information on Standard ML can be found in *ML for the Working Programmer* [8]. More information on HOL can be found in [12, 11, 10]. Unfortunately, these references describe an older version of HOL, known as HOL88, and there is no corresponding set of documentation describing HOL90. However, many of the differences between HOL88 and HOL90 are simple matters of syntax, so the HOL88 documentation is still a useful reference for HOL90. See [9] for a brief description of the differences. We hope that better documentation for HOL90 will become available in the near future.

Another useful resource is the X window application `xholhelp`, which provides online documentation of HOL theorems, functions, and tactics. It is distributed as part of the HOL90 source.

## Chapter 5

# Romulus Security Proofs

This chapter describes the Romulus process specifications and special-purpose Romulus tactics for proving processes secure, then finishes with a tutorial example.

The processes analyzed by Romulus for nondisclosure properties deal with multiple security levels (for example, unclassified, confidential, secret, and top secret). Romulus models these processes as rated state machines, which assign security levels to each input and output event. The raw information that can be obtained, or viewed, by an observer depends on that observer's security level.

Romulus includes support for showing restrictiveness of buffered server processes. A buffered process consists of a FIFO queue and a process being buffered; it saves its inputs on the queue until the process being buffered is ready to receive them. The process being buffered is a server process if it waits for input in a (possibly parameterized) state, processes each input by producing zero or more outputs, and then calls itself to again wait for input in a (possibly different parameterized) state.

### 5.1 Romulus Process Specifications

Full process specifications are given in Romulus using the Romulus Interface Process Specification Language (IPSL). An IPSL specification contains descriptions of a process's input and output ports, the messages passing through the ports, the process's response to these messages, and other information



needed to translate the specification to HOL. The full description of this language is given in section 5.1.1. IPSL uses the Romulus Process Specification Language (PSL) for the process description itself. PSL is described more fully in section 5.1.2.

IPSL specifications must be translated into HOL form before you can use HOL to reason about them. The command `ips12hol` translates process specifications written in IPSL into HOL90 process specifications. IPSL allows the user to write process specifications with minimal knowledge of HOL. Occasionally, it might be necessary to look directly at the HOL specification produced by the `ips12hol` command. For this reason, section 5.1.3 describes the output of the `ips12hol` command. Section 5.1.3 can be skipped without loss of continuity.

### 5.1.1 The Romulus Interface Process Specification Language

This section gives the complete description of the Romulus Interface Process Specification Language (IPSL).

IPSL consists of text strings labeled and delimited by keywords beginning with `??` and ending with `:`. The `??` combination is used because it is not used anywhere in input for HOL90 or the Standard ML compiler. In IPSL, text strings giving HOL variables, constants, and expressions need not, and must not, explicitly invoke the HOL90 term parser with `--` and `'--`; the translator will do this.

Some keywords are optional, others are required, and some must be used in combination with others when they are used. Some keywords must have associated text string values in order for `ips12hol` translations to succeed.

Process specifications describe either atomic server processes or composite processes. Atomic server process specifications are described first, followed by a description of composite process specifications. In these descriptions, a keyword is used to mean this keyword with its associated value, if any.

An atomic server process specification consists of the following:

- `??Process:` — This keyword gives the name of the process. If the process is not named, the IPSL translator creates a name for the process from its tree address.

- `??HOL_functions`: — This optional keyword introduces arbitrary HOL text that defines type or function constants used in the rest of the process's specification.
- Information about the security levels that the process's specification will use is optional. This information is given by the following keywords:
  - `??LevelTheory`: — the name of a HOL theory describing a set of levels and a dominance relation on these levels.
  - `??DomRelation`: — the name and HOL type of the dominance relation.
  - `??LevelVar`: — the name and HOL type of a variable denoting an arbitrary level.

If any of these three values is given, all must be, and each must have a value for the translation to succeed. If this information is not given, the translator uses standard default values.

- Information on the process's state parameter, if it has one, is optional. This information is given by the following keywords:
  - `??StateVar`: — a variable denoting an arbitrary state parameter.
  - `??Initial`: — the initial value of the state parameter.
  - `??Invariant`: — a predicate satisfied by all attainable values of the state parameter. The predicate is given as the right-hand side of an equation that defines the predicate as a function of the variable given by `??StateVar`. This keyword is optional; if it is not given, the predicate is taken to be T (i.e., the predicate that is always satisfied).
  - `??Projection`: — a function of a security level and a state parameter intended to give all, but only, the information in that state parameter that will influence the process's future behavior visible at that level. The function is given as the right-hand side of an equation that defines the function as a function of the process's `??LevelVar` value and the `??StateVar` value.

If any of these four values are given, all except `??Invariant` must be, and all that are given must have values for a translation to succeed.

- **??OutPort:** — This keyword gives the name of an output port and signals the start of an output port specification. If the output port is not named, the IPSL translator creates a name for the port from its tree address. At least one output port must be given for the translation to succeed. There must be one output port specification for each of the process's output ports.

The following are the contents of an output-port specification:

- **??MessageVar:** — There must be one or more of these values. The translator assumes that the format of messages through a port is given by a tuple of variables naming and giving the types of the entries in this tuple. Each **??MessageVar:** value gives one of these variables and its type.
- **??LevelFun:** — This function assigns security levels to arbitrary messages through the port. The function is given as the right-hand side of an equation defining the function in terms of the **??MessageVar:** variables.
- **??LevelRange:** — This keyword formally specifies the range of security levels of the messages passing through the port that must be proved to hold.

Everything about the output port, with the exception of its name, must be given for the translation process to succeed.

- **??InPort:** — This keyword gives the name of an input port and signals the start of an input port specification. If the input port is not named, the translator creates a name for the port from its tree address. At least one input port must be given for the translation to succeed. There must be one input port specification for each of the process's input ports.

The following are the contents of an input-port specification:

- **??MessageVar:** — There must be one or more of these values. The translator assumes that the format of messages through a port is given by a tuple of variables naming and giving the types of the entries in this tuple. Each **??MessageVar:** value gives one of these variables and its type.

- `??LevelFun`: — This function assigns security levels to arbitrary messages through the port. The function is given as the right-hand side of an equation defining the function in terms of the `??MessageVar`: variables.
- `??LevelRange`: — This keyword formally specifies the assumed range of security levels of the messages passing through the port.
- `??Response`: — This function gives the PSL process that the process being specified becomes in response to an arbitrary input message received through this port. This function is given as the right-hand side of an equation defining the function in terms of the `??MessageVar`: variables. The value associated with this keyword is a PSL process.

Everything about the input port, with the exception of its name, must be given for the translation process to succeed.

- `??EndProcess`: — The optional name value associated with this keyword is not used and is provided only to allow you to create more readable specifications.

A composite process specification consists of the following:

- `??Process`: — This keyword gives the name of the process. If the process is not named, the translator creates a name for the process from its tree address.
- `??HOL_functions`: — This optional keyword introduces arbitrary HOL text that defines type or function constants used in the rest of the process's specification. Constants that are used in two or more of the process's subprocesses must be declared here.
- Information about the security levels that the process's specification will use is optional. This information is given by the keywords `??LevelTheory`:, `??DomRelation`:, and `??LevelVar`: as described for atomic server processes. If any of these three values is given, all must be, and each must have a value for the translation to succeed. If this information is not given, the translator uses standard default values.

- **??OutPort:** — Output ports are specified as described for atomic server processes.
- **??InPort:** — Input ports are specified as described for atomic server processes except that no **??Response:** is specified.
- **??ProcessInFile:** — This keyword, used in the specification of a composite process, must be followed by the basename of an `.ips1` file containing the IPSL specification of a child process. There must be one of these keywords for each child. In typical cases, this keyword will be generated only by the graphical interface.
- **??Connection:** — This keyword, used in the specification of a composite process, must be followed by a pair of text strings each identifying a port on the process or one of its children. A port is identified by using its tree address. There must be one **??Connection:** keyword for each connection between a pair of ports. In typical cases, this keyword will be generated only by the graphical interface.
- **??EndProcess:** — The optional name value associated with this keyword is not used and is provided only to allow you to create more readable specifications.

An IPSL specification can be translated to a HOL specification using the graphical interface's `spec` command or the `ips12hol` translator. In either case, the IPSL specifications must be complete before they can be translated. An example of the output of the IPSL translator for an atomic process is given in section 5.1.3.

### 5.1.2 The Romulus Process Specification Language

The Romulus Process Specification Language (PSL) is used to specify processes. The syntax and an informal description of the semantics are given here; the formal semantics are given in [1] and in Volume II. A “process” is an abstraction of a state machine that describes its behavior solely in terms of input and output events.

PSL is a simple language with four basic processes—`Send`, `Receive`, `Call`, and `Skip`—and the means to combine these basic processes into more complex processes.

For technical reasons having to do with how HOL is defined, processes are defined in terms of *process-valued functions* and *invocations*. A process-valued function is a function that returns a process as its value. An invocation is a name for a process returned by an process-valued function. Later we will see how to map an invocation to the process it names.

Each of the basic processes is described below.

- The process **Send** transmits an output event; it takes a single argument:

*Send outevent*

The argument is an output event.

- The process **Receive** receives an input event; it takes two arguments:

*Receive select response*

The first argument, *select*, is a boolean predicate on input events; *select* determines which input events the **Receive** process responds to. The second, *response*, determines how the **Receive** process responds to each input event. An invocation constructor that maps input events to invocations is always used here.

- The process **Call** invokes other processes; it takes a single argument:

*Call process*

The *process* argument is an invocation that identifies the process to invoke. This invocation is always specified using an invocation constructor with the appropriate arguments, if any.

- The process **Skip** is the finished process that does nothing; it takes no arguments:

*Skip*

PSL has four operators for combining processes: the infix operator `;;`, `Orselect`, `If`, and `Buffered`. Each of these operators is described below.

- The infix operator `;;` is a sequence operator; it takes two operands:

*process\_1 ; ; process\_2*

Each operand is a process. The infix operator returns the process that consists of *process\_1* and *process\_2* executed in sequence.

- **Orselect** is a non-deterministic choice operator; it takes two operands:

*Orselect process\_1 process\_2*

Each operand is a process. **Orselect** returns the process that non-deterministically chooses to execute either *process\_1* or *process\_2*.

- **If** is the if-then-else operator; it takes three operands:

*If condition process\_1 process\_2*

The operands are a boolean predicate, *condition*, and two processes, *process\_1* and *process\_2*. **If** returns the process that executes *process\_1* if *condition* is true and executes *process\_2* otherwise.

- **Buffered** creates a process that buffers input events; it takes three operands:

*Buffered select buffer process*

The first operand, *select*, is a boolean predicate on input events; the second, *buffer* is a list of input events; and the third, *process*, is a process. **Buffered** returns the process that puts the input events satisfying the predicate into the buffer and passes them on to the process being buffered when that process is ready for them.

Parentheses may be used to make a PSL specification unambiguous. It is recommended that PSL specifications be fully parenthesized to avoid any ambiguity in the specification.

### 5.1.3 The HOL Embedding

IPSL is embedded in HOL by defining the appropriate types, functions, and constants for the specification. While the HOL overhead for a specification is imposing, the majority of it is straightforward and is automatically generated

by the `ips12hol` translator. The average Romulus user will rarely need to look directly at the HOL version of a process specification as the Romulus tactics make this largely unnecessary. This section can be skipped by most readers.

This section describes the HOL version of a process specification, that is, the HOL specification created by the `ips12hol` command. We use the `filter` process from Chapter 2 as an example. The IPSL specification for this example is given in Chapter 2. The HOL version of a process specification creates a HOL theory describing the process. In the following we describe each part of the file used to create this theory. For the `filter` process, this file is `filter.spec.sml`.

The HOL specification starts with a command that deletes old versions, if any, of the `filter` theory files. The next lines create a new theory called `filter`, load the necessary Romulus libraries, and put HOL in draft mode.

```
(* Remove any earlier versions of the theory. *)

System.Unsafe.SysIO.unlink "filter.holsig"
  handle e => print "no earlier filter.holsig to remove\n";
System.Unsafe.SysIO.unlink "filter.thms"
  handle e => print "no earlier filter.thms to remove\n";

(* Create the new theory in the Romulus environment. *)

new_theory "filter";
load_library_in_place(get_library_from_disk "romulus");
```

Next, the theory giving the global declarations for the whole system is made a parent of the current theory.

```
(* Parent globals theory. *)

new_parent "simple_example_globals";
```

This entry is added only if the translator is invoked on the top-level component `simple_example`, but not if the translator is invoked on `filter`.

The specification continues by making the `string` theory a parent of the current theory, and declaring a function `source_level`, which takes a source string and returns a security level.

```
(* filter-specific HOL and SML identifiers. *)
```



```

new_parent "string";
new_constant
  {Name="source_level",
   Ty == ':string->standard_level'==};

```

The ipsl2hol translator copies these lines directly from the ??HOL\_functions: section of the IPSL specification to the HOL specification.

The next section defines various types used in the specification. The ipsl2hol translator automatically generates these declarations.

```

(* Define filter types and/or SML identifiers for them. *)

val filterLevel = (ty_antiquity_of (--'level:standard_level'--));

```

The next part of the theory uses romcontype to define HOL types for output events, input events, and invocations. The function romcontype is a convenience function for easily declaring non-recursive constructed types of the form needed for Romulus specifications. It takes as input input a string that names a constructed type and a list of (string,(hol\_type)list) pairs that name the constructors for this type and give the types of their arguments. It returns the defining theorem for the newly defined type and the type of the new type.

The translator takes the port name and the names and types of message variables from the ??OutPort, ??InPort, and ??MessageVar fields of the IPSL specification. The translator automatically generates the invocations.

```

(* Define filter output and input event types. *)

val (filterOutEv_Def, filterOutEv) =
  romcontype
    "filterOutEv"
    [
      ("f_out",
       [
         (type_of(--'source:string'--)),
         (type_of(--'data:string'--))
       ])
    ]
);

val (filterInEv_Def, filterInEv) =
  romcontype

```

```

"filterInEv"
[
  ("f_in",
    [
      (type_of(--'source:string'--)),
      (type_of(--'data:string'--))
    ]
  )
];

(* Define filter invocations and PSL processes. *)

val (filterInvoc_Def, filterInvoc) =
romcontype
"filterInvoc"
[
  ("filterTop",      []),
  ("filterResponse", [==':~filterInEv'==])
];

```

Next, the specification defines SML identifiers that give the type of a filter process and the types of various functions defined later in the specification. The translator automatically generates these type definitions.

```

val filterProc =
  ty_antiq(==':(~filterOutEv,~filterInEv,~filterInvoc)process'==);

(* Define SML identifiers giving the names and types of all
additional filter functions to be defined. *)

val filterOutPred = --'filterOutPred:~filterOutEv -> bool'--;
val filterInPred = --'filterInPred:~filterInEv -> bool'--;
val filterOutLevel = --'filterOutLevel:~filterOutEv -> ~filterLevel'--;
val filterInLevel = --'filterInLevel:~filterInEv -> ~filterLevel'--;
val filtertop = --'filtertop:~filterProc'--;
val filterresponse = --'filterresponse:~filterInEv -> ~filterProc'--;
val filterInvocVal = --'filterInvocVal:~filterInvoc -> ~filterProc'--;

```

Next the specification defines the input and output predicates that check that levels assigned to input and output events are in the appropriate ranges. The translator takes the port name and the names and types of message variables from the ??OutPort:, ??InPort:, and ??MessageVar: fields of the IPSL specification. The level assignment is taken from the LevelFun: field and the upper and lower bounds of the level ranges are taken from the ??LevelRange: field of the IPSL specification.

```
(* Define filter predicates identifying output and input events. *)
```

```
new_recursive_definition {
name = "filterOutPred",
fixity = Prefix,
rec_axiom = filterOutEv_Def,
def =
--'
  (~filterOutPred
    (f_out
      (source:string)
      (data:string)
    ) =
    ((standard_dom)
      (unclassified)
      (unclassified)) /\
    ((standard_dom)
      (unclassified)
      (unclassified)))
'--};
```

```
new_recursive_definition {
name = "filterInPred",
fixity = Prefix,
rec_axiom = filterInEv_Def,
def =
--'
  (~filterInPred
    (f_in
      (source:string)
      (data:string)
    ) =
    ((standard_dom)
      (source_level source)
      (unclassified)) /\
    ((standard_dom)
      (top_secret)
      (source_level source)))
'--};
```

Next, the specification defines the functions that assign security levels to input and output events. The translator takes the port names and the names and types of message variables from the `??OutPort`, `??InPort`, and `??MessageVar` fields of the IPSL specification. The translator takes the functions's

definitions from the ??LevelFun fields of the IPSL specification.

```
(* Define filter functions assigning security levels to output and
input events. *)
```

```
new_recursive_definition {
name = "filterOutLevel",
fixity = Prefix,
rec_axiom = filterOutEv_Def,
def =
--'
  (~filterOutLevel
    (f_out
      (source:string)
      (data:string)
    ) = (unclassified))
--};
```

```
new_recursive_definition {
name = "filterInLevel",
fixity = Prefix,
rec_axiom = filterInEv_Def,
def =
--'
  (~filterInLevel
    (f_in
      (source:string)
      (data:string)
    ) = (source_level source))
--};
```

The specification defines the filter process in two parts: a process called `filtertop` and a process-valued function called `filterresponse`.<sup>1</sup> Each of these parts has a corresponding invocation constructor; these constructors are called `filterTop` and `filterResponse` respectively.

The translator automatically generates the definitions of `filtertop` and `filterresponse`. The translator also automatically generates the names `filtertop`, `filterTop`, `filterresponse`, and `filterResponse`.

```
(* Define the filter functions giving the top-level PSL process and
the response to input events. *)
```

---

<sup>1</sup>The specification defines `filtertop` as a process rather than a process-valued function because `filter` is a nonparameterized process.

```

new_definition(
  "filtertop",
  --'^filtertop =
  (Receive (\ev:~filterInEv. T) (filterResponse))
  '--);

```

The translator defines the process-valued function `filterresponse` using `new_recursive_definition` with the process definition taken from the `??Response`: field of the IPSL specification.

```

new_recursive_definition {
  name = "filterresponse",
  fixity = Prefix,
  rec_axiom = filterInEv_Def,
  def =
  --'
  (^filterresponse
    (f_in
      (source:string)
      (data:string)
    ) =
  ((If ((source_level source) = unclassified)
    (Send (f_out source data))
    Skip));
  (Call filterTop)))
  '--};

```

Next, the specification defines the function that maps the invocations to the processes they name. This function maps the invocation `filterTop` to the process `filtertop` and the invocation constructor `filterResponse` to the process-valued function `filterresponse`.

The translator automatically generates this mapping function.

```
(* Define the filter function assigning meanings to invocations. *)
```

```

new_recursive_definition {
  name = "filterInvocVal",
  fixity = Prefix,
  rec_axiom = filterInvoc_Def,
  def =
  --'
  (^filterInvocVal filterTop = filtertop) /\
  (^filterInvocVal (filterResponse ineq) = (filterresponse ineq))
  '--};

```

Finally, the specification exports the theory and exits from HOL.

```
export_theory();  
exit();
```

## 5.2 Proving Processes Secure

### 5.2.1 Trusted Processes

This section provides an overview of the Romulus tactics for proving that a buffered server process is restrictive and that the required interface conditions hold.

#### Restrictiveness

The raw information that can be obtained, or viewed, by an observer of a system depends on that observer's security level. What an observer at a given security level can know about a system is described by three functions: functions giving the security levels associated with input and output events, and a *projection function* hiding state information.

A projection function induces an equivalence partition on process state parameters, for the process being buffered, based on security level. Two parameters in the same equivalence partition with respect to one level must also be in the same equivalence partition with respect to any other level dominated by that level. The projection of a parameter to a level determines all the information, but only the information, necessary to determine the system's future behavior that can be viewed at that level.

The following are informal descriptions of conditions that are sufficient for guaranteeing, for buffered server processes, that an observer at a particular security level can never deduce anything about higher or incomparable-level events:

1. Any output produced in response to an input is at a level that dominates the level of that input.
2. If the level of an input is not dominated by the observer's level, then the projection function induces a state parameter partition such that

the process being buffered appears not to have changed state (i.e., the process's state after responding to this input is in the same partition block as it was before receiving this input).

3. If the security level of an input is dominated by that of the observer, then any two state parameters of the process being buffered that seemed equivalent to the observer are not distinguished by the response to this input:

- If any arbitrary choices made for one state parameter are made identically for the other state parameter, the sequences of PSL operations performed for the two parameters are the same.
- All outputs possible for one state parameter are also possible for the other.
- Any possible next state parameter for one parameter is in the same partition block as any possible next parameter for the other state parameter.
- The previous two conditions hold for any *combination* of visible outputs and seemingly equivalent next state parameters.

The buffering guarantees that the full process is always ready to accept input, so there is never any information conveyed about the process's state by its ability or inability to accept input. Note that conditions 2 and 3 are always satisfied by nonparameterized processes.

The predicates `BNPSP_restrictive` and `BPSP_restrictive` formalize these conditions for buffered, non-parameterized server processes and buffered, parameterized server processes respectively. Formal definitions of these conditions are given in Volume II of this documentation set.

## Using HOL to Prove Restrictiveness

The two main Romulus HOL predicates for expressing restrictiveness of PSL processes are `BPSP_restrictive` and `BNPSP_restrictive`, which apply to parameterized and non-parameterized processes, respectively. Both `BPSP_restrictive` and `BNPSP_restrictive` make additional assumptions beyond what is required to prove restrictiveness; they assume that the level of each input event arriving at an input port is in the level range for that

port. A process proved `BPSP_restrictive` or `BNPSP_restrictive` is restrictive, but only for the environment defined by the assumptions on the input events. `BPSP_restrictive` and `BNPSP_restrictive` also state the necessary interface conditions for the process. These conditions are that if the level of every input event arriving at an input port is in the level range for that port, then the level of every output event for every output port will be in the level range specified for that output port. Proving these interface conditions proves that the process conforms to the environment that is claimed for it in the graphical interface.

Each of these predicates is defined in terms of simpler predicates. Lists of these predicates and informal descriptions of what they mean follow. We give the much simpler nonparameterized case first.

The following predicates define `BNPSP_restrictive`:

- `BNPSP_rightform` is true of a nonparameterized process if that process is a server process (i.e., true if the process waits to receive an arbitrary input, processes this input, and calls itself to again wait for an arbitrary input).
- `BNPSP_nowritesdown` is true of a nonparameterized server process if all the outputs it produces in response to an arbitrary input event are at security levels that dominate the level of the input and if all the outputs it produces in response to an arbitrary input event are in the specified output ranges. The latter condition states the interface condition.

The following predicates define `BPSP_restrictive`:

- `BPSP_rightform` is true of a parameterized process if that process acts as a server process for any value of its state parameter that satisfies the user-supplied invariant (i.e., true if, when the process's state parameter satisfies the invariant, the process waits to receive an arbitrary input, processes this input, and calls itself to again wait for an arbitrary input with a possibly new value of its state parameter).
- `BPSP_invpreserved` is true of a parameterized process whose initial state parameter satisfies the invariant and that acts as a server process for any value of its parameter satisfying the invariant if its response to any input event ends with calling itself with a value of its state parameter that also satisfies the invariant.



- `BPSP_nowritesdown` is true of a parameterized server process if all the outputs it produces in response to an arbitrary input event are at security levels that dominate the level of the input and if all the outputs it produces in response to an arbitrary input event are in the specified output ranges. The latter condition states the interface condition.
- `BPSP_nolowchange` is true of a parameterized server process whose state parameter always satisfies the invariant, and a user-supplied projection function whose value for a security level and a state parameter is intended to capture just the information in the state parameter that influences the process's future behavior that is visible at that security level, if for every input not visible at a particular level the process's state parameter seems not to change — that is, if an input is not visible at a level, the projection to that level of the process's state parameter in its call to itself after responding to the input is the same as the projection to that level of the process's state parameter before receiving the input.
- `BPSP_samepath` is true of a parameterized process and projection function satisfying `BPSP_nolowchange` if for any level, any two state parameters whose projections to this level are equal, and any input visible at this level, if the process' responses to this input when parameterized by these parameters make the same nondeterministic choices then they execute the same sequence of PSL commands.
- `BPSP_lowresponsesame` is true of a parameterized process and projection function satisfying `BPSP_nolowchange` if the process's responses to every input that is visible at a particular level does not distinguish any two possible values of its state parameter that seem equivalent at that level — that is, for any two possible values of the state parameter whose projections to a level are the same, the responses to an input visible at that level produce the same outputs visible at that level and produce seemingly equal new state parameters.

Each of these security predicates has a corresponding tactic in the Romulus library that automates much or all of the effort of proving that the predicate holds. The tactic corresponding to each Romulus security predicate

is named by appending `_TAC` to the name of the predicate: `BPSP_restrictive_TAC`, `BNPSP_restrictive_TAC`, and so on. These tactics each do two or more of the following things:

- They expand the definitions of the predicates they correspond to.
- They apply other Romulus tactics that can be expected to automatically solve or simplify the subgoals they generate. For instance, `BNPSP_restrictive_TAC` automatically applies `BNPSP_rightform_TAC` and `BNPSP_nowritesdown_TAC`.
- They perform case splits on all possible types of input events (essentially, on each input port).
- They apply Romulus-specific theorems about security predicates and PSL processes as rewrite rules until all specific mention of security predicates and PSL processes goes away. For instance, being a server process is defined in terms of responding to any input event by ending with a call to itself to wait for the next input. Ending with a call to itself is defined by finishing all processing before making a call to itself. A process `P1 ; ; P2` finishes all processing if and only if `P1` finishes all processing and `P2` finishes all processing, and a `Send` or `Skip` PSL process always finishes all processing.
- They expand definitions that the user will almost certainly need to expand to prove any remaining subgoals (e.g., the definition of the dominance relation on security levels for the `BPSP_nowritesdown` and `BNPSP_nowritesdown` goals).

It is important that the various parts of showing restrictiveness (e.g., showing `BPSP_nolowchange` and `BPSP_lowresponsesame`) can be stated as separate goals and proved as separate theorems. This separation avoids wasteful repetition when proving restrictiveness for large, complicated processes. It is also important that the subgoals left by the various Romulus tactics are usually all of the same form, so the `THEN` tactical can often be used to solve several of these subgoals at the same time.

A simple example of using the Romulus tactics to prove restrictiveness can be found in Chapter 2.

### 5.2.2 Manifestly Secure Processes

This section provides an overview of the Romulus tactic for proving that the necessary conditions hold for manifestly secure processes.

A process whose outputs are all at the same or higher levels than its inputs is assumed to be manifestly secure by the flow analyzer because there is no way for high level information to leak to low levels in such a process. The security of such processes depends entirely on the levels assigned to its inputs and to its outputs, not on what the process actually does.

The conditions that must hold for a manifestly secure process are that

- the level of every output event for a port is in the level range claimed for that port, and
- if the level of every input event for a port is in the level range for that port, then the level of every output event dominates the level of every input event.

For atomic processes whose specifications have at least one input port for which no `Response` function is specified, the IPSL translator generates a goal file whose goal states these manifest security conditions for the process.

Note that this goal is generated even if the level ranges on the process's ports do not indicate that the process is manifestly secure. The process may be manifestly secure if the level assignment functions are taken into consideration, but this can be proved only by using HOL and cannot be recognized by the graphical interface.

Romulus provides a tactic that automates much or all of the effort of proving that the manifest security conditions hold for a process. This tactic, `ManifestlySecureTAC`, makes case splits on input and output events, and then rewrites with the definitions of the input and output predicates, the input and output level-assignment functions, and the dominance relation on levels. In typical cases, it reduces this goal to showing that the levels of output events dominate the levels of input events.

### 5.2.3 Composite Processes

This section provides an overview of the Romulus tactic for proving that the components of composite process are properly connected.

Composite processes are composed of subprocesses whose components are connected together. These connections must ensure that the levels of events assigned by the sending port are consistent with the levels assigned these events by the receiving port. The checks that must be made to ensure this consistency are that

- the level range of the sending port is a subrange of the level range of the receiving port, and
- the level assigned to an event by the sending port is the same as the level assigned to the event by the receiving port.

For composite processes, the IPSL translator always generates a goal file whose goal states connection conditions for each connection.

Romulus provides a tactic that automates much or all of the effort of proving that all the connection conditions hold for a composite process. This tactic, `HookupValidTAC`, splits the goal into subgoals and then rewrites with the definitions of the input and output predicates. For validly connected processes, the conditions that must be proved are usually trivial, and `HookupValidTAC` typically proves the goal automatically.

### 5.3 Tutorial Example

This section contains a somewhat more realistic example than the simple example given in Chapter 2 of how Romulus can be used to examine nondisclosure security for a system. The example in this section is a simple secure version of a token ring local area network (LAN). In section 5.3.1 we describe how token rings work and how a secure version can work. In section 5.3.2 we describe how the Romulus graphical interface can be used to give a top-level decomposition of the architecture of a single station on the LAN. In section 5.3.3 we show how the Romulus graphical flow analyzer can be used to analyze the architecture to identify security problems and to determine which components need to be trusted. In sections 5.3.4 and 5.3.5 we give the specification of one of the trusted components of the token ring using IPSL. In section 5.3.6 we give the proof in HOL of the security condition for this component. In section 5.3.7 we consider the specification and proof of a single-level trusted component. Last, in section 5.3.8 we show how we confirm our results using the graphical interface.

### 5.3.1 Secure Token Ring Station

A token ring LAN consists of a number of stations linked together in a circle. The main purpose of the token ring protocol is to control access to the transmission medium so that transmissions from different stations do not interfere with each other.

Access is controlled by means of a unique message called the *token*. At initialization, a token is sent by some station. This token starts out traveling in a certain direction around the ring. A station receiving the token does one of two things:

- If the station has messages to send, it begins sending them in the direction that the token was going. Stations between the sender and the intended destination receive and retransmit these messages. When these messages reach the intended receiver, it sends them to the local host and also retransmits them in the direction in which they were traveling. When these messages reach the original sender, it transmits the token in the same direction in which the token was originally circulating.
- If the station has no messages to send, it retransmits the token in the direction the token was traveling when received.

This protocol ensures (modulo noise, dropped messages, etc.) that at most one station's messages are being transmitted at a time.

In the secure version of the token ring that we examine in the remainder of this tutorial, each host on the ring handles a single security level of data, with possibly many different levels of host on the network. Thus, the hosts are single-level, but the network itself is multilevel. We assume that all stations on the ring know what level is associated with each host, and that messages on the ring (other than the token) are assigned the level of the station where they originated. Mediation is done at the receiving end; that is, when a message arrives at a station and the message header identifies that station as the destination, then the station checks to see that the originator's level is less than or equal to that of the station.

### 5.3.2 Graphical Interface

In this section we describe the graphical decomposition of a single station on the secure token ring LAN; this station is attached to a host handling **secret** data. The decomposition is shown in Figure 5.1. This figure shows use of the `names` top-level graphical interface command or the `portnamesdisplayed` interface parameter to display all port names.

#### The Station as a Whole

The station as a whole is represented as a component whose bounding box is almost the entire picture; this component is labeled `token_ring_station` in the upper-left corner. The station has four ports to its environment. The input ports are represented as circles and the output ports as diamonds. All ports and components in this example have been given descriptive names.

On the left side of the box, there is an input port labeled `t_in`. This is the port through which messages enter the station. As displayed in the picture, `t_in` has associated with it a range of security levels reflecting the range of levels of events that can pass through this port. Since all traffic on the ring goes through every station, this port must allow events of all levels to pass through it. The level range of `t_in` is therefore all levels from **unclassified** to **top\_secret**, which are abbreviated by `U` and `TS` respectively.

On the right side of the box, there is an output port labeled `t_out`. This is the port through which messages leave the station. Again, its level range is `[U,TS]`.

On the bottom of the box, there is an input port labeled `from_host`. This is the port through which messages from the station's host enter the station. Since the station is single level **secret**, the level range on this port is from **secret** to **secret** (abbreviated by `[S,S]`), that is, only **secret** messages can pass through this port.

On the top of the box, there is an output port labeled `to_host`. This is the port through which messages from the station enter the station's host. Again, the level range on this port is `[S,S]`.

#### The Sorter

The arrow from `t_in` indicates that it is connected to port `s_in`, which is an input port of the component `sorter`. This component splits the stream of

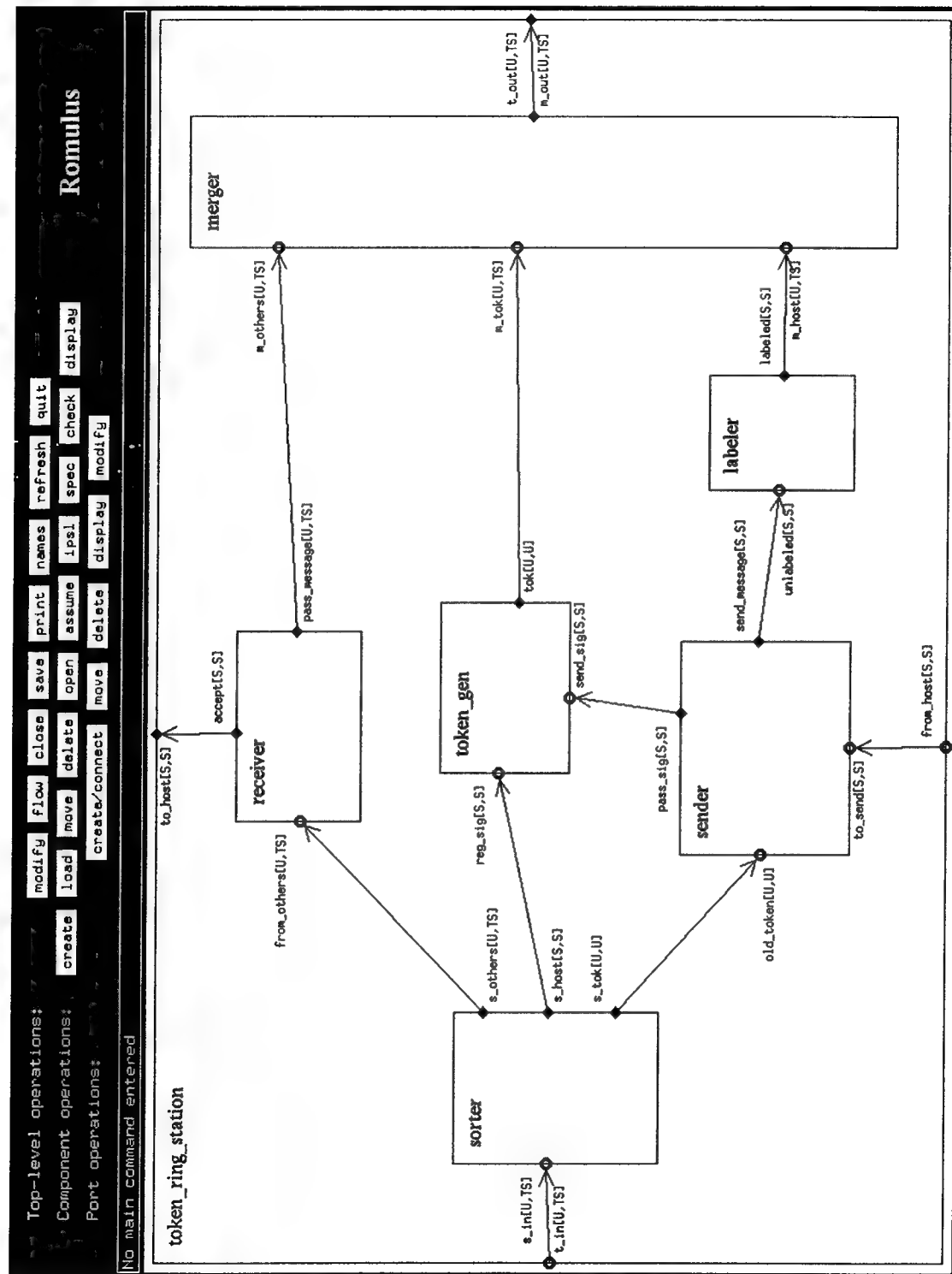


Figure 5.1: Token ring station

incoming messages into three internal streams:

1. tokens, which produce signals that go out through port `s_tok` to the sender component;
2. messages from this station (presumably returning from a trip around the ring), which produce signals that go out port `s_host` to the `token_gen` component; and
3. messages from other stations, which go out port `s_others` to the receiver component.

### The Sender

The sender component receives messages from the attached host through the ports `from_host` and `to_send`. When it has a message to send, it waits for a signal through the port `old_token`. This signal indicates that the station has the token. When and if it does get this signal, it sends the message out `send_message` to the `labeler` component. If it gets this signal and does not have a message to send, it sends a signal out port `pass_sig` to `token_gen` to generate a token to go out on the LAN (that is, to pass the token).

### The Receiver

The receiver component gets messages from other stations through port `from_others`. If the message is for this station, receiver checks to see that the level of the originating station is less than or equal to that of this station. If it is, receiver strips off the header and sends the message to the host through ports `accept` and `to_host`. Whether or not the message is for this station, and whether or not it passed mediation, the receiver sends the message out through port `pass_message` to be passed on (this is so that it may return to the originating station, which will then regenerate the token).

### The Labeler

The labeler component gets messages from the host via sender through port `unlabeled`. It attaches header information, including the security relevant information of the originating host, and sends the result out through port `labeled`.



## The Token Generator

The token generator, `token_gen`, gets signals from the `sender` through port `send_sig` and from the `sorter` through port `reg_sig`. Its response is to create a token and send it out port `tok` for transmission on the ring.

## The Merger

The `merger` component takes three streams of messages (messages from other stations being passed on around the ring through `m_others`, tokens through `m_tok`, and messages from the attached host through `m_host`) and merges the three streams into a single stream that is sent out on the LAN medium through ports `m_out` and `t_out`.

### 5.3.3 Flow Analysis

Repeated use of the flow analyzer and the component `assume` command, as described in Chapter 2, reveals that every subcomponent except `sender` and `labeler` must be assumed or proved secure in order to have the security of the full station follow from the security of its parts. Later we will see that the `labeler` must also be secure, even though this fact is not revealed by flow analysis.

For the `sorter` component, for example, multilevel information comes in through the `sorter` and can pass through the token generator to the `merger` and then out on the LAN. The reason this flow is found to be insecure is that there is nothing, in the absence of a more detailed analysis of the `sorter`, to ensure that the multilevel data coming in is not mixed or insecurely relabeled before it is passed on. Note that the graphical interface arbitrarily shows only one of the three possibly insecure flows through the `sorter`.

The `receiver` and `merger` similarly could take messages in at one level and relabel them to go out on the network at another level, or pass on messages to the host that the host is not authorized to receive. A more detailed specification of each is necessary. The `labeler` must be proved secure to make sure that messages are correctly labeled, but this is not revealed by flow analysis.

For the `token_gen` component, we have `secret` signals entering `token_gen` through `reg_sig` and `send_sig`. These signals cause the generation

of a token, which is unclassified. The `token_gen` component appears to be downgrading `secret` information. We cannot prove this component restrictive since it is not. The Romulus analysis identifies this component as a place where there are potential security problems. Other analysis techniques, not provided by Romulus, can be used to determine the extent of the problem. In many cases, security engineering techniques such as channel bandwidth reduction are adequate to convince the analyst that the potential problem is not an actual security vulnerability. In this particular example, the token ring stations do not allow the token to be directly visible to hosts, thus severely reducing the channel bandwidth.

### 5.3.4 Sorter Specification and Proof

We next show how to use Romulus to prepare a formal specification of the `sorter` process and use standard Romulus tactics to prove in HOL90 that it is secure.

#### Overview

We first give an informal English description of the `sorter` process, the data types of the pieces of information going to and from it, and the level assignments we make for this data.

**Functionality** The `sorter` takes inputs from a single port connected to a token ring LAN. Each message, except the token message, includes information about the message's origin and destination. If a message is identified as being "the token", none of its other components are defined. The `sorter` takes the messages it receives and sorts them into three groups: the token, messages from the station on which the `sorter` is running, and messages from other stations.

**Being Nonparameterized** The `sorter` processes each input as it arrives and does not retain any information about an input after it has been processed. The `sorter` is thus a memoryless process and can be taken to be a nonparameterized server process. Proving security is significantly simpler for nonparameterized processes than parameterized ones, since possible transfers of information via the process's state parameter need not be considered.

**Messages** We take full messages as containing the following four pieces of information: a flag identifying whether the message is the token; the message's source and destination; and the data in the message.

**Inputs and Outputs** The sorter takes inputs from the port to the LAN medium and generates outputs through the three ports corresponding to the three groups of messages. It takes in full messages from `s_in` and sends out full messages to `s_others`. It sends out simple signals, which we formalize as objects of the HOL type `one`, to acknowledge receipt of a message originally sent by the host station or receipt of a token. (The HOL type `one` contains one object, and the only thing known about this object is that it is the only object of its type.)

**Level Assignments** We assign the level `unclassified` to full messages that are tokens and assign other messages the levels of their senders, which we take to be given by a HOL constant `station_level`, whose type is a function that maps names of stations to their levels.

### 5.3.5 IPSL Specification

We use the Romulus graphical interface's `Top-level operations save command` to produce an IPSL specification of each of the token ring processes. We edit the file `sorter.ipsl` to produce the following IPSL specification:

```
??Process: sorter

??OutPort: s_host
??MessageVar: x:one
??LevelFun: station_level this_station
??LevelRange: secret secret

??OutPort: s_others
??MessageVar: tokenflag:bool
??MessageVar: from:string
??MessageVar: to:string
??MessageVar: data:(num)list
??LevelFun: tokenflag => unclassified | (station_level from)
??LevelRange: unclassified top_secret
```

```

??OutPort: s_tok
??MessageVar: x:one
??LevelFun: unclassified
??LevelRange: unclassified unclassified

??InPort: s_in
??MessageVar: tokenflag:bool
??MessageVar: from:string
??MessageVar: to:string
??MessageVar: data:(num)list
??LevelFun: tokenflag => unclassified | (station_level from)
??LevelRange: unclassified top_secret
??Response:
(If tokenflag
  (Send (s_tok one))
  (If (from = this_station)
    (Send (s_host one))
    (Send (s_others tokenflag from to data)))));
(Call sorterTop)

??EndProcess: sorter

```

The globals file for this example, `token_ring_station_globals.sml`, makes the theory string a parent of the current theory to define the HOL type `:string`, defines `station_level` as a function that takes a string naming a station and returns the level of that station, defines `this_station` as an unspecified string naming the current station, and declares an axiom that says that `station_level this_station` is `secret`.

The `sorter` specification treats the `from` and `to` components of messages as character strings and the `data` component of messages as a list of numbers, though this can easily be generalized by taking these things to be of variable types.

The output port `s_host` outputs signals of type `:one`, and these signals have the level of the current station. The output port `s_others` outputs full messages, and these messages have the level of their sender, if they are not tokens. The output port `s_tok` outputs signals of type `:one`, and these signals have the level `unclassified`.

The input port `s_in` defines the `sorter`'s response to all inputs. If a message is a token, the `sorter` sends a signal out of `s_tok`. Otherwise, if the message is from the current station, the sender sends a signal out of `s_host`.

Otherwise, the sorter sends the full message out of `s_others`. Input messages are assigned the level `unclassified` if they are tokens and are assigned the level of their source otherwise.

### 5.3.6 Showing Security Using HOL

We now describe in detail how to prove in HOL90 that the sorter is restrictive.

First, we prepare formal HOL90 specifications of the sorter component and the goal that must be proved about it to show that it is restrictive by running the translator `ips12hol` on the `.ips1` file for the top-level process:

```
ips12hol token_ring_station
```

The translator takes the `.ips1` extension as being implicit and adds it automatically. The translator produces three files of interest, `sorter.spec.sml`, `sorter.goal.sml`, and `token_ring_station_globals.sml`, as well as specification and goal files for each of the other processes. The top-level component `token_ring_station` is translated here so that the correct reference to the `token_ring_station_globals` theory is placed in the file `sorter.spec.sml`.

Next, we translate the global definitions file

```
rh01 <token_ring_station_globals.sml
```

The HOL theory produced by this file is a parent of each of the HOL theories for each component.

Next, we give the file `sorter.spec.sml` as input to HOL90 to produce a HOL theory of the sorter process. We use the version of HOL that has the Romulus library preloaded with the command

```
rh01 < sorter.spec.sml
```

We note here, though, that using this technique is appropriate only after the `.ips1` file has been debugged. Before then, it is more useful to run HOL90, give the specification file produced by `ips12hol` to HOL90 with the `use` command (as described below in constructing the proof of security), and have HOL90 available so that one can see error messages and experiment to find the problem.

The goal file produced by the translator for the sorter contains the following lines:

```

(* Load the Romulus library and the theory sorter. *)

let
  val romulus_lib = find_library "romulus";
in
  load_theory "sorter";
  load_library_in_place romulus_lib
end;

(* Using standard security levels in the Romulus library. *)

(* Define SML identifiers giving names and types for possibly
polymorphic sorter functions and predicates. *)

val sorterOutPred = romgetconstant "sorterOutPred";
val sorterInPred = romgetconstant "sorterInPred";
val sorterOutLevel = romgetconstant "sorterOutLevel";
val sorterInLevel = romgetconstant "sorterInLevel";
val sorterInvocVal = romgetconstant "sorterInvocVal";
val sorterTop = romgetconstant "sorterTop";

(* Set the sorter restrictiveness goal. *)

g('BNPSP_restrictive
  ^sorterInPred
  ^sorterOutPred
  (standard_dom)
  ^sorterInLevel
  ^sorterOutLevel
  ^sorterInvocVal
  ^sorterTop');

(* DO PROOFS.  PROBABLE BEST FIRST STEP: e(BNPSP_restrictive_TAC); *)

(* Save the result and close with a call to romrtheory to communicate
the result to the Romulus graphical interface. *)

(* UNCOMMENT THESE LINES WHEN YOU FINISH THE PROOF
save_top_thm("sorter_BNPSP_restrictive");
romrtheory("sorter");
export_theory();
exit();
END OF COMMENTED-OUT LINES *)

```

The translator-produced comments in this file explain most of the lines in it, give a suggestion as to how to produce the proof, and contain the final lines one should use after the proof is completed to save the result proved, invoke the Romulus utility `romrtheory` to produce a file for communicating the result proved back to the Romulus graphical interface, and save the sorter theory.

We note here that the utility `romgetconstant` addresses a problem that does not arise in the current formulation of the sorter example: a polymorphic constant must be given an explicit type, though that type can be given by a type variable, before it can be used in a goal. The utility `romgetconstant` finds appropriate instantiations of possibly polymorphic constants.

Note that all the lines after setting the goal are comments, so that giving this file as input to HOL90 leaves HOL90 ready to prove the security goal. We run the Romulus version of HOL90 with the command

```
rh01
```

producing the usual HOL90 banner and prompt.

At the `-` prompt, we give the goal file as input to the prover with the command:

```
use "sorter.goal.sml";
```

producing the following response:

```
[opening sorter.goal.sml]
```

```
Loading theory "sorter"
```

```
Theory "rom_temp" exported.
```

```
The library "romulus" is already loaded.
```

```
val it = () : unit
```

```
val sorterOutPred = (--'sorterOutPred'--): term
```

```
val sorterInPred = (--'sorterInPred'--): term
```

```
val sorterOutLevel = (--'sorterOutLevel'--): term
```

```
val sorterInLevel = (--'sorterInLevel'--): term
```

```
val sorterInvocVal = (--'sorterInvocVal'--): term
```

```
val sorterTop = (--'sorterTop'--): term
```

```
(--'BNPSP_restrictive sorterInPred sorterOutPred standard_dom
  sorterInLevel
  sorterOutLevel
```

```

        sorterInvocVal
        sorterTop'--)
=====

```

```

val it = () : unit
val it = () : unit

```

This response confirms that the Romulus library is loaded and sets the goal of showing that the sorter is `BNPSP_restrictive`, since its inputs are taken to be buffered and it is a nonparameterized server process. The goal asserts that the `BNPSP_restrictive` relation holds for the predicates defining the input and output level ranges, for the standard dominance relation on security levels, for the functions assigning security levels to the sorter's input and output events, for the function assigning meanings to names used in defining the sorter process, and for one of these names, `sorterTop`, which names the full sorter process. (Another of these names gives the response of the sorter to input events; the `ipsl2hol` translator typically generates all needed uses of such response names in the HOL specification files it produces.)

Taking the translator-recommended first step in producing the proof with the command

```
e(BNPSP_restrictive_TAC);
```

produces the following response:

```

OK..
4 subgoals:
(--'standard_dom (station_level from) (station_level from)'--)
=====
  (--'~tokenflag'--)
  (--'~(from = this_station)'--)
  (--'standard_dom (station_level from) unclassified'--)
  (--'standard_dom top_secret (station_level from)'--)

(--'standard_dom
  (station_level this_station) (station_level this_station)'--)
=====
  (--'~tokenflag'--)
  (--'from = this_station'--)

```



```

(--'standard_dom (station_level this_station) unclassified'--)
(--'standard_dom top_secret (station_level this_station)'--)

(--'standard_dom secret (station_level this_station)'--)
=====
(--'tokenflag'--)
(--'from = this_station'--)
(--'standard_dom (station_level this_station) unclassified'--)
(--'standard_dom top_secret (station_level this_station)'--)

(--'standard_dom (station_level this_station) secret'--)
=====
(--'tokenflag'--)
(--'from = this_station'--)
(--'standard_dom (station_level this_station) unclassified'--)
(--'standard_dom top_secret (station_level this_station)'--)

val it = () : unit

```

The tactic `BNPSP_restrictive_TAC` expands the definition of `BNPSP_restrictive`, does case splits on possible input events, applies rewrite rules that simplify away references to PSL processes, and rewrites with the definitions of the functions assigning security levels to input and output events. Four simple subgoals remain.

The first subgoal

```

(--'standard_dom (station_level this_station) secret'--)
=====
(--'tokenflag'--)
(--'from = this_station'--)
(--'standard_dom (station_level this_station) unclassified'--)
(--'standard_dom top_secret (station_level this_station)'--)

```

asserts that the standard level of this station dominates the level `secret`. But this is true, of course, simply because the level of this station is `secret`. We rewrite this goal using this fact.

```

e(REWRITE_TAC
  [axiom "token_ring_station_globals" "station_is_secret"]);

```

This produces the following response:

```

OK..
1 subgoal:
(--'standard_dom secret secret'--)
=====
  (--'tokenflag'--)
  (--'from = this_station'--)
  (--'standard_dom (station_level this_station) unclassified'--)
  (--'standard_dom top_secret (station_level this_station)'--)

```

```

val it = () : unit

```

This subgoal asserts that the standard level **secret** dominates itself. This is true, of course, simply because every standard level dominates itself. The Romulus library theory **romlemmas** contains a theorem **standard\_dom\_reflexive** making this assertion, so the command

```

e(MATCH_ACCEPT_TAC (theorem "romlemmas" "standard_dom_reflexive"));

```

produces the response

```

OK..

```

```

Goal proved.
|- standard_dom secret secret

```

```

Goal proved.
|- standard_dom (station_level this_station) secret

```

```

Remaining subgoals:
(--'standard_dom (station_level from) (station_level from)'--)
=====
  (--'tokenflag'--)
  (--'from = this_station'--)
  (--'standard_dom (station_level from) unclassified'--)
  (--'standard_dom top_secret (station_level from)'--)

```

```

(--'standard_dom
  (station_level this_station) (station_level this_station)'--)
=====
  (--'tokenflag'--)
  (--'from = this_station'--)
  (--'standard_dom (station_level this_station) unclassified'--)

```

```

(--'standard_dom top_secret (station_level this_station)')

```

```

(--'standard_dom secret (station_level this_station)')
=====
(--'tokenflag')
(--'from = this_station')
(--'standard_dom (station_level this_station) unclassified')
(--'standard_dom top_secret (station_level this_station)')

val it = () : unit

```

We could produce separate proofs of each of the remaining subgoals, but a quick glance shows that *all* the four subgoals can be solved in the same way. The HOL THEN tactical automatically applies a tactic to all remaining subgoals, so rather than go on we back up and use THEN on the original goal. We back up to the original goal by using the command

```
b();
```

three times, then give the command

```

e(BNPSP_restrictive_TAC THEN
  REWRITE_TAC
    [axiom "token_ring_station_globals" "station_is_secret",
     theorem "romlemmas" "standard_dom_reflexive"]);

```

which gives the response

```
OK..
```

```
Goal proved.
```

```

|- BNPSP_restrictive sorterInPred sorterOutPred standard_dom
   sorterInLevel
   sorterOutLevel
   sorterInvocVal
   sorterTop

```

```
Top goal proved.
```

```
val it = () : unit
```

We have thus now proved that the sorter is restrictive.

### 5.3.7 Labeler Specification and Proof

The labeler must also be proven secure, although this becomes apparent through an inability to prove the manifest security conditions for the process and not from flow analysis. Consider the following scenario. The labeler's inputs and outputs are expected to be secret, so flow analysis assumes that this process is manifestly secure. This would lead you to complete the specification of this process as a manifestly secure process. The specification for the **labeler**, as a manifestly secure process, is

```
??Process: labeler

??OutPort: labeled.
??MessageVar: tokenflag:bool
??MessageVar: from:string
??MessageVar: to:string
??MessageVar: data:(num)list
??LevelFun: tokenflag => unclassified | (station_level from)
??LevelRange: secret secret

??InPort: unlabeled
??MessageVar: to:string
??MessageVar: data:(num)list
??LevelFun: station_level this_station
??LevelRange: secret secret
??Response:

??EndProcess: labeler
```

In this specification, no **Response:** is provided. The goal file for this specification sets the goal of proving the manifest security conditions for the labeler. Unfortunately, attempts to prove this goal will fail because they reduce to proving that

```
standard_dom secret (station_level from)
```

and

```
standard_dom (station_level from) secret
```

where *from* is the *from* field of the outgoing message. The specification does not specify *how* this field is to be filled in so it is impossible to draw any conclusions about it.

The specification of the labeler must contain additional information. This is done by completing the specification by filling in the `??Response:` field and proving that the labeler is trusted. The complete specification is

```

??Process: labeler

??OutPort: labeled
??MessageVar: tokenflag:bool
??MessageVar: from:string
??MessageVar: to:string
??MessageVar: data:(num)list
??LevelFun: tokenflag => unclassified | (station_level from)
??LevelRange: secret secret

??InPort: unlabeled
??MessageVar: to:string
??MessageVar: data:(num)list
??LevelFun: station_level this_station
??LevelRange: secret secret
??Response:
(Send (labeled F this_station to data));;
(Call labelerTop)

??EndProcess: labeler

```

This specification specifies that the `from` field of the outgoing message is `this_station`, which has level `secret`.

The goal for this specification is to prove that it is `BNPSP_restrictive`; the proof is similar to the proof of the sorter.

### 5.3.8 Confirming Security in the Graphical Interface

We next save the proof of security, save the theorem that the sorter is secure, and produce a file for communicating this result to the Romulus graphical interface.

To save the proof, we rename `sorter.goal.sml` to `sorter.proof.sml` and edit its final lines to

```

e(BNPSP_restrictive_TAC THEN
  REWRITE_TAC
  [axiom "token_ring_station_globals" "station_is_secret",
   theorem "romlemmas" "standard_dom_reflexive"]);

```

```
(* Save the result and close with a call to romrtheory to communicate
the result to the Romulus graphical interface. *)
```

```
save_top_thm("sorter_BNPSP_restrictive");
romrtheory("sorter");
export_theory();
exit();
```

(It is often convenient to do this step earlier, as the proof is being developed, so the proof file can be used to record partial proofs.)

Executing the final four lines, gives the following response:

```
val it =
  |- BNPSP_restrictive sorterInPred sorterOutPred standard_dom
    sorterInLevel sorterOutLevel sorterInvocVal sorterTop  : thm

val it = () : unit
-
Theory "sorter" exported.
val it = () : unit
```

followed by exiting from HOL.

The call to `romrtheory` produces the file `sorter.rth`. This file notes that the parents of the `sorter` theory are the theories in the Romulus library, that the theory `sorter` does not define any security levels or any dominance relations on them, that appropriate bounds have been proved on the levels of messages going in or out of any of the sorter's ports, that the sorter's ports are distinct, and that the sorter has been proved to be restrictive.

We can use the graphical interface to confirm that the sorter has been proved secure by starting it with the command

```
romulus -initial=tokenring
```

We select the graphical interface's component check command and then select the sorter component by clicking on it with the left mouse button. The file `sorter.rth` is consulted and, if everything checks out, the proof of security will be confirmed by placing a double asterisk in the lower-right corner of the sorter component.

If you get the following error:

```
unable to open theory file romlemmas.rth
```

you must produce rtheory files for the theories in the Romulus library. These files can be produced with the command

```
echo 'romrtheory "";' | rhof
```

This command will create the appropriate rtheory files in the current directory. If desired, these rtheory files can be placed in a central directory and this directory can be added to the directories listed using the searchpath resource.

## Chapter 6

# The Authentication Protocol Toolkit

This chapter describes how to use the Romulus implementation of the logic of authentication that is presented in Volume II of the Romulus documentation set. First, in this section we outline the steps of using the tool. Next, we give the mechanics of using the tool to describe and specify a protocol and to derive a proof. Then, we provide a tutorial example. We recommend reading the entire chapter before trying to use the authentication protocol toolkit.

While this tool is practical for proving protocols correct, it is the first version of the tool, and the interface is still primitive. We intend, in a later version, to provide an interface that enables you to relate the correctness proof to what is achieved at intermediate stages of the protocol execution. We will also provide proof support that will make most proving automatic. An ideal implementation would use a front-end language that would make it unnecessary for the user to deal with HOL. However, for someone doing detailed analysis of protocols, the small amount of work needed to deal with HOL is not significant.

A convenient technique for constructing HOL theories and proofs is to open two windows, a HOL window and a text window. The HOL window runs HOL and the text window is used to edit your theory or proof file. HOL commands are first entered into the text window as a permanent record of the session and then copied into the HOL window to try them out. When you are finished, the text file you created can be used to recreate the HOL theory or proof. In this chapter, we describe only the text files.



Here are the general steps for using the authentication logic implemented in HOL. We invoke the tool by first invoking a version of HOL that has the Romulus library preloaded. We use the command

```
rhof
```

from a UNIX shell interface. (If you are unfamiliar with HOL, refer to Chapter 4.)

The protocol to be analyzed must be described and specified (recall that we use the word “describe” to mean defining the protocol message sequence, which is like giving code at a high level). The next step, then, is to create an SML file that contains this information; we describe this step in section 6.1. We then proceed with the analysis by loading the information from the SML file into HOL. For example, if the information about the protocol is contained in the file `foo.sml`, then it is loaded into HOL by typing the command

```
use "foo.sml";
```

at the HOL prompt. Loading the SML file loads in the initial assumptions, as well the messages, and defines the desired final position, which will later be the goal to be proved. We can then use the full power of HOL to attempt to prove the goals. We currently use a few standard HOL tactics for this task, although we expect to almost completely automate this proof process in the future.

## 6.1 Describing and Specifying a Protocol

In this section, we explain the mechanics of describing and specifying a protocol by showing how we would use it on the Denning-Sacco authentication protocol. We handle this example more fully in section 6.3.

We commence building a new theory, in this case called `ds_90`, in an SML file called say, `ds_90.sml`, with the command:

```
new_theory("ds_90");
```

Next, we declare the basic objects used in this particular protocol.

```
new_constant{Name = "A",  
              Ty  == ':principal'==};  
new_constant{Name = "Svr",
```

```

        Ty == ':principal'==};
new_constant{Name = "Kas",
        Ty == '~textlist'==};
...

```

In this case, two principals A and Svr are declared along with a key, Kas, used to encrypt messages between them.

The order in which we describe the protocol and specify it does not matter. Suppose we specify it first. We declare the initial assumptions, one of which is shown here.

```

(* Preconditions *)
new_open_axiom("dsa1",
    --'theorem(believes A (is_shared_secret A Svr Kas))'--);
...

```

This axiom declares that A believes that the key Kas is a shared secret between A and Svr.

We next declare the things we want to be true at the end of protocol execution. All of the constants involved will have been declared already, though we have not shown all of them. The following is a sample goal, and not necessarily the usual thing one wants to prove of a protocol.

```

new_definition ("postcond", --'postcondition =
    theorem(possesses A Kab) /\
    theorem(believes A (convey Svr ((name B) APP Kab APP Ts))) /\
    theorem(believes A (is_fresh ((name B) APP Kab APP Ts))) /\
    theorem(possesses B Kab) /\
    theorem(believes B (convey Svr ((name A) APP Kab APP Ts))) /\
    theorem(believes B (is_fresh ((name A) APP Kab APP Ts)))'--);

```

While we have given the initial assumptions as separate facts (axioms), we make the postcondition a single statement. This is so that we have a standard way of (later) presenting the criterion for correctness as a single proof goal. The definition gives the name `postcondition` to the postcondition. We will later make `postcondition` the goal of a HOL proof session.

Finally, we describe the protocol in HOL form.

```

(* The messages *)
new_open_axiom("dsm1", --'send A Svr ((name A) APP (name B))'--);
new_open_axiom("dsm2", --'send Svr A (encrypt Kas ((name B) APP Kab
    APP Ts APP (encrypt Kbs ((name A) APP Kab APP Ts))))'--);
new_open_axiom("dsm3", --'send A B
    (encrypt Kbs ((name A) APP Kab APP Ts))'--);

```

The theory is completed, and we save it, close it, and exit from HOL.

```
export_theory();  
close_theory();  
exit();
```

## 6.2 Deriving a Proof

We open a new SML file, called say, `ds_proof_90.sml`. We will build the proof in this file.

We start by opening a new theory, loading the theory describing the protocol, and declaring the HOL proof goal.

```
new_theory("ds_proof_90");  
new_parent "ds_90";  
set_goal([], --'postcondition'--);
```

Now it is a matter of carrying out a HOL proof. Our first step should be to expand `postcondition`. The HOL tactic

```
e(PURE_ONCE_REWRITE_TAC [definition "ds_90" "postcond"]);
```

will achieve this step. The goal is now the conjunction of six smaller statements, which are pulled off and proved one by one in standard HOL style.

Certain facts are needed multiple times, and a sensible proof will first prove these facts before proving the postcondition. We will call such a fact a “lemma”, though to HOL it is simply a theorem and has the same status as any other theorem. These lemmas—proved theorems—are then adduced as needed during the proof. As is usual with theorem proving, we cannot know in advance what lemmas are needed. And so during the proof, we sometimes suspend work on our current goal, stop and state a lemma, and make it a new goal and prove it. After proving and saving the lemma, we can continue with the original goal.

In fact, instead of making the postcondition the initial goal, it is sensible first to prove the six conjuncts separately, as lemmas. Then, when we make the postcondition our goal, it is quickly proved by calling upon the six lemmas. In the Romulus library of models (Volume III), we treat the Needham-Schroeder protocol in just this way.

Suppose we decide to prove, as a lemma, the first conjunct.

```
set_goal([], --'theorem(possesses A Kab)'--);
```

Through examination of the logical rules, we find a sensible axiom to apply at this stage (in fact it is axiom P4, which says that possession of a part of message follows from possession of the whole). Continuing, we eventually reduce the proof obligations (subgoals) to known facts (most commonly those axioms in the theory "ds\_90").

Note that the method shown here is rather primitive. We intend to develop more advanced methods to facilitate users. For example, it would be desirable to write a more complicated tactic to repeatedly do

```
CONJ_TAC, ACCEPT_TAC, MATCH_ACCEPT_TAC, MATCH_MP_TAC
```

which is a common pattern of arriving at proofs.

## 6.3 Tutorial Example

Here, we go over sufficient parts of the analysis of the Denning-Sacco protocol to demonstrate the use of the tool to the new user. After this tutorial, you should be able to analyze protocols on your own. For further reference, see Volume III of this documentation set; that volume covers this protocol, and also the Needham-Schroeder protocol, in greater depth. The full proof transcripts are presented there.

From the previous sections, you should be familiar with the mechanics of using the tool. When we discuss the contents of the theories and proofs, therefore, we will not usually point out in what files they belong.

The Denning-Sacco protocol is typically presented in the literature as follows.

1. A → Svr: A, B;
2. Svr → A : {B, Kab, Ts, {A, Kab, Ts}\_e(Kbs)}\_e(Kas);
3. A → B : {A, Kab, Ts}\_e(Kbs);

We begin with an informal discussion of how the protocol works. It is assumed that A and B each has a key (Kas and Kbs, respectively) with which it can communicate securely with the server, Svr. A wishes to establish a new key to share with B, so that they may have a session using encryption. A sends a message, consisting of its and B's names, in the clear, to the server, the

intent of which is understood by the server. The server sends the new key  $K_{ab}$  to A in a “certificate”, containing a timestamp  $T_s$ . The timestamp ensures A that this message is fresh. The server signs the certificate by encrypting it with  $K_{as}$ , which is shared by it and A. This encryption also assures that no one else can read or tamper with the contents. The server avoids having to communicate with B by passing this responsibility to A: in its message to A, the server includes an analogous certificate for B. A (and anyone else other than Svr or B) is unable to read or alter the contents of this certificate. In the third message, A passes this certificate on to B.

This protocol is efficient and simple, and it is commonly used. The use of timestamps is adequate to ensure freshness if process clocks are tightly synchronized. However, the protocol does not have a “handshake” at the end, and so A and B cannot be sure that the other has successfully received the key.

We must express the protocol in a form suitable for the machinery implemented in HOL. We start with declarations of the various objects in the protocol and then give the protocol itself. You may wish to look first at the protocol definition and then at the declarations of the objects.

```
(* The principals and necessary objects.*)
new_constant{Name = "A",
              Ty == ':principal'==};
new_constant{Name = "B",
              Ty == ':principal'==};
new_constant{Name = "Svr",
              Ty == ':principal'==};
new_constant{Name = "Ts",
              Ty == ':~textlist'==};
new_constant{Name = "Kas",
              Ty == ':~textlist'==};
new_constant{Name = "Kbs",
              Ty == ':~textlist'==};
new_constant{Name = "Kab",
              Ty == ':~textlist'==};
```

Now, it is simple to write this protocol in the HOL form:

```
(* The messages sent between principals*)
new_open_axiom("dsm1", --'send A Svr ((name A) APP (name B))'--);
new_open_axiom("dsm2",
               --'send Svr A (encrypt Kas ((name B) APP Kab APP Ts APP
```

```

                (encrypt Kbs ((name A) APP Kab APP Ts))))'--);
new_open_axiom("dsm3",
                --'send A B (encrypt Kbs ((name A) APP Kab APP Ts))'--);

```

Next we enter the initial assumptions. These assumptions break naturally into three groups — one for each principal. Here are the assumptions for Principal A. We do not show those for B and Svr.

```

(* Preconditions for A *)
new_open_axiom("dsa1",
                --'theorem(believes A (is_shared_secret A Svr Kas))'--);
new_open_axiom("dsa2", --'theorem(believes A (is_fresh Ts))'--);
new_open_axiom("dsa3", --'theorem(believes A (is_recog (name B)))'--);
new_open_axiom("dsa4", --'theorem(possesses A Kas)'--);
new_open_axiom("dsa5", --'theorem(possesses A (name A))'--);
new_open_axiom("dsa6", --'theorem(possesses A (name B))'--);

(* Preconditions for B *)
...

(* Preconditions for Svr *)
...

```

It is hard, if not impossible, to determine all the preconditions before beginning the proof. As with ordinary program development, it is natural to develop specifications, the executable, and the correctness proof or argument in concert. These initial conditions are enriched or modified as the proof progresses, and the user discovers assumptions he or she needs to make (in this context, the tool is functioning as an analysis tool).

For the final conditions that we wish to hold after protocol execution, we will choose something very simple. For a more complete specification, see the library of models (Volume III).

```

new_definition ("postcond", --'postcondition =
                theorem(possesses A Kab)'--);

```

In the proof session, we define this as the goal, and then plan our proof strategy. We examine the messages of the protocol and the axioms of authentication logic. We see that A possesses the key Kab as a result of its receiving the message dsm2, which has the contents:

```

encrypt Kas ((name B) APP Kab APP Ts APP
              (encrypt Kbs ((name A) APP Kab APP Ts))).

```

A gets this message and possesses the key Kas, which it shares with the server Svr. A can decrypt the message and extract Kab from the concatenated objects in the list. There are axioms that reflect all of these actions. We first use axiom P4

```

!p x y. theorem(possesses p(x APP y)) ==> theorem(possesses p y)

```

to reduce the proof obligation to

```

theorem(possesses A ((name B) APP Kab))

```

and then the analogous axiom P3 to produce the subgoal

```

theorem(possesses A ((name B) APP Kab APP Ts APP
                      (encrypt Kbs ((name A) APP Kab APP Ts)))).

```

Continuing in the obvious fashion, we go on to use the relevant axioms to reduce this subgoal to a two subgoals: that A possesses an encrypted copy of this textlist, and that A possesses the encryption key. The former follows ultimately from the fact that A was sent the message, while the latter is an initial assumption.

It is now a standard exercise in theorem proving, in the HOL theorem prover. Each of the subgoals generated is ultimately proved true, where the last step is usually showing that the subgoal is an initial axiom. For more details, follow the treatment of this protocol in the Romulus library of models (Volume III). Each step of the proof is in the proof transcript, and so the proof can be examined one step at a time.

## Appendix A

# Graphics Interface Parameters

As noted earlier, Romulus allows the user to set several parameters that control things such as how information is displayed, and it supplies reasonable default values for these parameters if the user does not set them. More precisely, these parameters are X windows *application resource values*, since they are values assigned to resource variables associated with Romulus as a whole rather than with particular widgets inside Romulus.

Romulus is a customizable X application whose application resource values can be set in various defaults files or with command-line arguments. The Romulus-specific application resource values can also be set in environment variables. Further, Romulus is customizable for some standard X application resource values, such as *geometry*, that are not Romulus-specific.

If a Romulus application resource value is set in more than one way, Romulus uses the value with the highest priority. Romulus assigns priorities to the different possibilities, with the first highest, as follows:

1. Command-line arguments.
2. Environment variables.
3. X defaults files in the priority order given by the windowing system and version of the X Toolkit Intrinsics being used. (See [6] for a discussion of the different possibilities and their priorities.)
4. Romulus-supplied defaults.



The only aspect of this ordering that is specific to Romulus is giving environment variables a priority lower than command-line arguments but higher than any other source for resource values.

The mechanisms Romulus uses for setting application resource values with defaults files or command-line arguments are standard for an X application; see [6] for a thorough discussion of these mechanisms. This chapter gives the additional information needed to set each Romulus-specific resource variable in each of three different ways: in an X defaults file; with an environment variable; or with a command-line argument. It also gives the default values Romulus assigns to each of these resource variables if they are not otherwise assigned values. The following general considerations apply in setting all Romulus application resources.

Every Romulus-specific application resource value is either a text string, a nonnegative integer, or a boolean. All of these values can be specified in X defaults files, environment variables, or command line arguments as unquoted text strings. Integers can be given in ordinary base-10 notation, and booleans can be given with the strings `True` and `False`. Romulus and the X Toolkit Intrinsics automatically perform all necessary type conversions.

## A.1 X Defaults Files

Values can be set in X defaults files in many different ways; see [6]. One way, requiring no special privileges, is setting the environment variable `XENVIRONMENT` to the full pathname of a file, then setting values in this file. Another way, typically requiring system operator privileges, is creating a file named `Romulus`, after the class name of all Romulus applications, in the application-defaults directory for the operating system being used. On UNIX operating systems, this directory is usually named `/usr/lib/X11/app-defaults`. Defaults in `XENVIRONMENT` files have higher priorities than defaults in application-defaults files.

The unabbreviated syntax for specifying an application resource in an X defaults file is

*application\_name.resource\_name: value*

The application name for an executable version of Romulus is the name of the file containing it, typically `romulus`. So, for example, the application resource `geometry` can be set in an X defaults file with the line

```
romulus.geometry: 1000x500+0+0
```

This syntax is used for application resources that are not specific to Romulus-type applications and that apply to only one executable version of Romulus.

For an application resource whose name identifies it as being unique to a particular type of application and whose value is to be used for all applications of that type, the unabbreviated syntax can be abbreviated to

```
*resource_name: value
```

Thus, a Romulus-specific application resource such as `initial` can be set to the string `token_ring` in an X defaults file with the line

```
*initial: token_ring
```

There are other possibilities, and some forms for assigning values have higher priorities than others; see [6].

## A.2 Environment Variables

Every Romulus-specific application resource has a corresponding environment variable, which can be set with the usual UNIX `setenv` command. For example, to set the application resource `initial` to the string `token_ring` with an environment variable, for example, one uses

```
setenv ROM_INITIAL token_ring
```

Romulus environment variable names are typically shorter than the corresponding application resource names to reduce the amount of typing needed.

## A.3 Command-Line Options

Every Romulus-specific application resource has a corresponding command line option. All of these command-line options begin with a hyphen (-), end with an equals sign (=), and are “sticky,” meaning that the text string giving the value to be set immediately follows the option. For example, to set the application resource `initial` to the string `token_ring` with a command-line argument, for example, one can use the command line

```
romulus -initial=token_ring
```

to start Romulus. The command-line options are uniformly generated from the corresponding environment variable names by removing the initial `ROM_`,

changing upper case to lower case, prepending an initial -, and appending a final =. Be careful about escaping wildcard characters in your shell.

## A.4 Romulus-Specific Application Resources

The remaining sections of this chapter describe the Romulus-specific X application resources, which we refer to simply as “resources.” Each section names a resource with the name it is given in an X defaults file, gives an English description of it, gives its default value, and gives the environment variable and command-line option corresponding to it. Default font names are given in full, but user-chosen font names can be given either in full, via aliases, or with wildcards; once again, see [6]. Be careful about escaping wildcards characters in your shell.

### A.4.1 abbreviations

The resource `abbreviations` gives abbreviations for the names of security levels. Romulus uses these abbreviations in its displays of security-level ranges associated with ports and as the labels on buttons for selecting security levels. The resource is a text string consisting of equations of the form

*name = abbreviation*

separated by new-line characters. In setting this resource, two characteristics shared by the C programming language, the UNIX operating system, and the X11 resource processing tools are convenient: a new-line character is given by `\n`; and a carriage return preceded by a backslash (`\`) is ignored, so that long strings can be given on multiple lines.

Default:

```
systemlow = Lo\n\  
unclassified = U\n\  
confidential = C\n\  
secret = S\n\  
top_secret = TS\n\  
systemhigh = Hi
```

Environment variable: ROM\_ABBREVIATIONS  
Command-line option: `-abbreviations=`

#### A.4.2 abortsavefile

The resource `abortsavefile` names the file that Romulus produces containing a saved-component representation of the system under study if Romulus aborts. The format of this file is the same as if it had been produced with the top-level `save` command.

Default: `SAVED_SYSTEM_ON_ABORT`

Environment variable: `ROM_ABORTSAVEFILE`

Command-line option: `-abortsavefile=`

#### A.4.3 arrowheadrise

Romulus uses two numbers to determine the size and shape of the arrowheads that show the direction of data flow. Each arrowhead is the same size and fits in an isosceles triangle whose perpendicular bisector is the “shaft” of the arrow. Half the length of this triangle’s base is the arrowhead’s *rise*, and the triangle’s height, measured along the perpendicular bisector, is the arrowhead’s *run*. The resource `arrowheadrise` is the arrowhead’s rise, measured in pixels.

Default: 7

Environment variable: `ROM_ARISE`

Command-line option: `-arise=`

#### A.4.4 arrowheadrun

Romulus uses two numbers to determine the size and shape of the arrowheads that show the direction of data flow. Each arrowhead is the same size and fits in an isosceles triangle whose perpendicular bisector is the “shaft” of the arrow. Half the length of this triangle’s base is the arrowhead’s *rise*, and the triangle’s height, measured along the perpendicular bisector, is the arrowhead’s *run*. The resource `arrowheadrun` is the arrowhead’s run, measured in pixels.

Default: 14

Environment variable: `ROM_ARUN`

Command-line option: `-arun=`

#### A.4.5 `buttonfont`

The resource `buttonfont` names the font used for command buttons.

Default:

`-misc-fixed-medium-r-normal--13-120-75-75-c80-iso8859-1`

Environment variable: `ROM_BF`

Command-line option: `-bf=`

#### A.4.6 `comerroredge`

When Romulus detects a possibly insecure data-flow path with the `flow` command, it displays this path by emphasizing the arrows, icons, and boxes representing the connections, ports, and components that are involved in the path. The resource `comerroredge` is the thickness, in pixels, of an emphasized box's walls.

Default: 3

Environment variable: `ROM_CERROR`

Command-line option: `-cerror=`

#### A.4.7 `comnormedge`

The resource `comnormedge` gives the thickness, in pixels, of the walls of the boxes Romulus displays to represent components that are not involved in a path showing a possibly insecure data flow detected by the `flow` command. Default: 0. A thickness of 0 is treated as a thickness of 1, but is drawn faster by some X servers.

Environment variable: `ROM_CNORM`

Command-line option: `-cnorm=`

#### A.4.8 `componentfont`

Romulus represents a component as a box containing that component's name or its tree address in parentheses if it is unnamed. The resource `componentfont` names the font used for displaying these names.

Default:

`-adobe-times-medium-r-normal--18-180-75-75-p-94-iso8859-1`

Environment variable: `ROM_CF`

Command-line option: `-cf=`

#### A.4.9 `editfont`

The resource `editfont` names the font used in the edit windows that appear when the top-level `modify` command is used.

Default:

`-misc-fixed-medium-r-normal--15-140-75-75-c-90-iso8859-1`

Environment variable: `ROM_EF`

Command-line option: `-ef=`

#### A.4.10 `initial`

The resource `initial` gives a basename that together with the extension `.rom` names a saved-component file. Romulus makes the component saved in this file into the main component initially being studied.

Default: the empty string, which Romulus interprets as an empty component

Environment variable: `ROM_INITIAL`

Command-line option: `-initial=`

#### A.4.11 `labelfont`

The resource `labelfont` names the font used to label the rows of command buttons and also used in the message window.

Default:

`-misc-fixed-medium-r-normal--15-140-75-75-c-90-iso8859-1`

Environment variable: `ROM_LF`

Command-line option: `-lf=`

#### A.4.12 `levelfile`

The resource `levelfile` gives a basename that together with the extension `.rth` names an rtheory file. If this file exists and is a valid rtheory file, Romulus takes the security levels named and assigned an order in this file as the set

of possible security levels. Otherwise, Romulus takes the possible security levels as `systemlow`, `unclassified`, `confidential`, `secret`, `top_secret`, and `systemhigh`, in that order.

Default: `levels.input`

Environment variable: `ROM.LEVELFILE`

Command-line option: `-levelfile=`

### A.4.13 logofont

The resource `logofont` names the font used to display the “Romulus” logo when this logo has not been replaced by one of the text-entry windows.

Default:

`-adobe-times-bold-r-normal--24-240-75-75-p-132-iso8859-1`

Environment variable: `ROM.LOGOF`

Command-line option: `-logof=`

### A.4.14 mineditwidth

When the top-level `modify` command is active, selecting a component or port on the canvas causes an edit window to appear in which the user can modify all of the text strings associated with that component or port. These edit windows automatically resize themselves to fit the strings displayed if the user makes them longer. The windows only expand to the right, though, and cannot *reposition* themselves to prevent their right edges from going outside the canvas and disappearing. Romulus thus positions these edit windows so that they can become at least a minimum number of characters wide before their right edges go off the canvas. The resource `mineditwidth` is this number of characters.

Default: 35

Environment variable: `ROM.MINW`

Command-line option: `-minw=`

### A.4.15 porterrorline

When Romulus detects a possibly insecure data-flow path with the `flow` command, it displays this path by emphasizing the arrows, icons, and boxes representing the connections, ports, and components that are involved in the

path. The resource `porterrorline` is the thickness, in pixels, of the lines in an emphasized arrow's shaft and head.

Default: 3

Environment variable: `ROM_PERROR`

Command-line option: `-perror=`

#### **A.4.16 portfont**

The resource `portfont` names the font used for port names or port tree addresses when they are displayed. The display of port names is controlled with the top-level `names` command and the `portnamesdisplayed` resource.

Default:

`-misc-fixed-medium-r-semicondensed--13-120-75-75-c60-iso8859-1`

Environment variable: `ROM_PF`

Command-line option: `-pf=`

#### **A.4.17 portnamesdisplayed**

The resource `portnamesdisplayed` determines whether Romulus initially displays port names.

Default: False

Environment variable: `ROM_PNAMES`

Command-line option: `-pnames=`

#### **A.4.18 portnormline**

The resource `portnormline` gives the thickness, in pixels, of the lines in the shaft and head of the arrows Romulus draws to represent connections between ports that are not involved in a path showing a possibly insecure data flow detected by the `flow` command.

Default: 0. A thickness of 0 is treated as a thickness of 1, but is drawn faster by some X servers.

Environment variable: `ROM_PNORM`

Command-line option: `-pnorm=`



#### **A.4.19 searchpath**

When searching for saved-component or rtheory files, Romulus looks in the current directory and in the directories given by the **searchpath** resource. The values of this resource are character strings containing directory names separated by colons.

Default: the empty string, which Romulus interprets by searching only in the current directory

Environment variable: ROM\_SEARCHPATH

Command-line option: **-searchpath=**

#### **A.4.20 textbuttonfont**

The text-entry windows that appear when the top-level **save**, the component **load**, and the port **modify** commands are active contain subcommand buttons that confirm text entry or identify the use to be made of the text. The **textbuttonfont** resource names the font used on these buttons.

Default:

**-misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso8859-1**

Environment variable: ROM\_TBF

Command-line option: **-tbf=**

#### **A.4.21 texteditfont**

The text-entry windows that appear when the top-level **save**, the component **load**, and the port **modify** commands are active contain an edit window into which all keyboard input is directed. The **textbuttonfont** resource names the font used in this edit window.

Default:

**-misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso8859-1**

Environment variable: ROM\_TEF

Command-line option: **-tef=**

#### **A.4.22 textlabelfont**

The text-entry windows contain labels that identify these windows. The **textlabelfont** resource names the font used for these labels.

Default:

`-misc-fixed-medium-r-semicondensed--13-120-75-75-c-60-iso8859-1`

Environment variable: `ROM_TLF`

Command-line option: `-tlf=`

#### A.4.23 translationstable

The edit windows produced with the top-level `modify` command are similar to invocations of an Emacs-style editor. Editing changes can be confirmed, which corresponds to saving the file being edited, or canceled, which corresponds to exiting the editor without saving any changes, with particular combinations of keystrokes. (They can also be confirmed or canceled with mouse-button clicks, as described earlier for the top-level `modify` command.)

The resource `translationstable` is a character string defining which combinations of keystrokes invoke the actions of confirming or canceling all changes. See [6], under `XtParseTranslationTable`, for a description of the format of such strings and their interpretations.

Default:

```
Ctrl<Key>X, Ctrl<Key>S: confirmedit() \n\  
Ctrl<Key>X, Ctrl<Key>C: canceledit()
```

This default causes changes to be confirmed by the key sequence `control-X control-S` and canceled by the key sequence `control-X control-C`.

Environment variable: `ROM_TRANSLATIONSTABLE`

Command-line option: `-translationstable=`

## Appendix B

# The Romulus HOL90 Library

The Romulus HOL90 library provides an environment for specifying and proving security and correctness properties of processes and protocols, and for communicating proof results to the Romulus graphical interface. It contains the Romulus theories giving HOL90 formalizations of processes, process nondisclosure security, protocols, and protocol correctness. It contains utilities useful for specifying processes and setting goals of proving these processes secure, and a utility for producing files that communicate selected contents of HOL90 theories to the Romulus graphical interface. It also contains tactics that greatly aid in producing proofs that processes have particular nondisclosure properties.

This appendix describes the Romulus HOL90 library and its contents. It first describes HOL90 libraries in general and the aspects of using them that arise most frequently in Romulus. It then describes the individual elements of the Romulus HOL90 library.

### B.1 HOL90 Libraries

A HOL90 library is intended to provide an environment for doing HOL work. A library is given by a group of files on disk, a group of files arranged in a particular way, together with identifying and initialization information.

The files for a library must be contained in a directory with three sub-directories named `help`, `src`, and `theories`. The `help` directory contains on-line help information for the library. The `src` directory contains func-

tions (typically utilities and tactics) that are loaded with the library, and the `theories` directory contains the theories that are loaded with the library.

The `theories` subdirectory must itself contain subdirectories `src` and `ascii`. The `src` directory contains the Standard ML of New Jersey (SML) source code that defines these theories in HOL90, and the `ascii` directory contains the ASCII text files that HOL90 produces to store these theories on disk. (In the future, HOL90 will be able to store theories using binary formats appropriate to particular machines, in which case the `theories` directory will also contain subdirectories such as `sun4` containing these binary files.)

The *path* of a library is the absolute pathname of the directory containing the `help`, `src`, and `theories` subdirectories for the library. This path, information identifying the `src` functions and theories to load when the library is loaded, information giving the order in which the functions and theories are to be loaded, and information giving any additional code to be executed after the functions and theories are loaded, are given by a `.hol.lib` file for the library. Every `.hol.lib` file has a corresponding HOL90 internal representation as an SML object of type `lib`.

All access to a library in HOL90 is through that library's `lib` object. The function `find_library` takes a library's name, and if the library has not already been loaded searches for its `.hol.lib` file in the directories given by the HOL90 global `library_path`. If it finds the library's `.hol.lib` file, `find_library` computes and returns the corresponding `lib` object.

The function `load_library` loads the library given by a `lib` object. Loading a library basically consists of loading the functions associated with the library and making the theories associated with the library into parent theories of the current theory. There is a subtlety, though: new theories cannot be added as parents of the current theory unless the current theory is in draft mode. The function `load_library` thus takes an additional string argument naming a new theory that it will create and make into the new current theory if the current theory is not already in draft mode and does not already have the library's theories as parents. If this string argument is `"-"`, meaning "the current theory", it is ignored.

In a situation where loading a library will not need to create a new theory (e.g., where the current library is already in draft mode, or already has all the library's theories as parents), one can use the function `load_library_in-place`. It takes a `lib` value as its only argument.

There is an additional subtlety in the case where one wishes to load a

theory having a library's theories as parents, leave that theory as the current theory, and leave HOL90 in proof mode. The following lines do this for theory `foo` and library `bar`:

```
let
  val bar_lib = find_library "bar";
in
  load_theory "foo";
  load_library_in_place bar_lib
end;
```

This does the following things:

1. make the `bar` library known to the system, which includes putting the paths to its theories on the HOL90 global `theory_path`;
2. load the theory `foo` (which will work because the paths to the theories of `bar` are now on `theory_path`) and makes the needed theories of `bar` into parent theories of `foo`;
3. load the `bar` library (which does not require the system to be put into draft mode because the theories of the library are already ancestors of the current theory).

This technique is used in all the goal files produced by the Romulus `ips12hol` translator.

The remainder of this appendix describes the contents of the Romulus library on a subdirectory by subdirectory basis and describes the code that is executed after the `src` functions and theories are loaded. For further general information on HOL90 libraries, see the file `doc/library.doc` in the HOL90 source directory.

## B.2 On-line Help Files

On-line help files are not yet available for the Romulus library. The Romulus library subdirectory `help` is empty.

## B.3 Utilities and Tactics

The Romulus library subdirectory `src` contains signature and SML source files for the following functions. The signature files, with suffix `.sig`, give the SML types of these functions, and the source files, with suffix `.sml`, give their definitions.

The functions `romcontype` and `romrecord` are utilities provided for user convenience in specifying processes. They are given by the files `romdeftypes.sig` and `romdeftypes.sml`.

The function `romgetconstant` is a utility provided for user convenience in setting possibly polymorphic goals. It is given by the files `romgetconstant.sig` and `romgetconstant.sml`.

The function `romrtheory` is a utility for communicating HOL90 results to the Romulus graphical interface. It is given by the files `romrtheory.sig` and `romrtheory.sml`.

The tactics `BNPSP_rightform_TAC`, `BNPSP_nowritesdown_TAC`, `BNPSP_restrictive_TAC`, `BPSP_rightform_TAC`, `BPSP_invpreserved_TAC`, `BPSP_nowritesdown_TAC`, `BPSP_nolowchange_TAC`, `BPSP_samepath_TAC`, `BPSP_lowresponsesame_TAC`, and `BPSP_restrictive_TAC` are tactics for proving restrictiveness, or some part of restrictiveness, for buffered, nonparameterized server processes (BNPSP) or buffered, parameterized server processes (BPSP). The tactics `ManifestlySecure_TAC` and `HookupValid_TAC` are tactics for proving the necessary conditions for manifestly secure and compound processes. These tactics are found in the files `romtactics.sig` and `romtactics.sml`.

Detailed descriptions of each of these functions follow. The descriptions of the tactics contain frequent references to “PSL processes”, which are members of the concrete recursive type PSL defined in the Romulus library theory `romproc`.

### B.3.1 `romcontype`

Function `romcontype` is a convenience for easily defining non-recursive concrete recursive types of the form needed for Romulus specifications.<sup>1</sup> This function takes as input a string naming the type to be defined, and a list of

---

<sup>1</sup>This function is needed for HOL90, Release 5 only.

(string, (hol\_type)list) pairs naming the constructors for this type and giving the types of their arguments. It calls HOL90 function `dtype` with this information, naming the defining theorem for the type by appending `_Def` to the type name, assuming that all constructors are prefix, and applying the function `Hty` to identify the types as HOL types. It returns a pair consisting of the defining theorem for the newly-defined type, and the type of the new type with variable types instantiated by the same type variables used to declare them.

### B.3.2 romrecord

Function `romrecord` defines what are effectively arbitrary record types. It takes as input a string naming a record type and a list of (string, hol\_type) pairs giving the record type's entries and their types. It defines a concrete recursive type for this record with `Make_` followed by the record type name as its sole constructor, and defines access and update functions for each entry in the record. It gives the access function for the entry named `<entryname>` the name `<entryname>`, and gives the update function for this entry the name `update_<entryname>`.

(Note that entries of different records in the same theory must have different names.) It returns a pair consisting of the defining theorem for the new record and the type of the new record with variable-type entries instantiated by the same type variables used to declare them.

### B.3.3 romgetconstant

Function `romgetconstant` is a simple utility for obtaining terms which are possibly-polymorphic constants with their unconstrained type variables instantiated to the same type variables used to declare them. It is useful for setting possibly polymorphic goals and determining whether a process is parameterized.

### B.3.4 romrtheory

Function `romrtheory` produces exactly the `rtheory` files needed to have the Romulus graphics interface identify the standard security levels and their order and to have it recognize that atomic processes have been proved secure,

manifest security conditions have been proved or that composite processes are properly connected. It assumes that the contents of the Romulus library are known. It determines which that no bounds have been assumed or proved on the security levels of messages into or out of any port, and assumes that the only security condition that must be checked is having a theorem of a standardized name asserting an appropriate security property for the process. If the function is called with an empty string it produces the rtheory files for the theories in the Romulus library.

### **B.3.5 BNPSP\_rightform\_TAC**

BNPSP\_rightform\_TAC simplifies a goal involving BNPSP\_rightform, which asserts that a PSL process is a non-parameterized server process, meaning that the end of its response to each input is to call itself to wait for the next input. It does case splits on possible inputs and applies rewrites that inductively define termination and calling oneself (Terminates and Loops-back) for PSL processes. In typical cases, it automatically proves the goal.

### **B.3.6 BNPSP\_nowritesdown\_TAC**

BNPSP\_nowritesdown\_TAC simplifies a goal involving BNPSP\_nowritesdown, which asserts that a non-parameterized server process does not produce any outputs in response to an input that are at security levels not dominated by the security level of the input or that are not at levels in the specified ranges. It does case splits on possible inputs and applies rewrites that inductively define "not writing down" (NoWritesDown) for PSL processes. In typical cases, it reduces this goal to showing that the levels of output events are dominated by the levels of input events.

### **B.3.7 BNPSP\_restrictive\_TAC**

BNPSP\_restrictive\_TAC is the main Romulus tactic for proving buffered, non-parameterized server processes restrictive. It takes the goal, rewrites it to expand out the definition of BNPSP\_restrictive, and applies BNPSP\_rightform\_TAC and BNPSP\_nowritesdown\_TAC to the two main subgoals.



### **B.3.8 BPSP\_rightform\_TAC**

BPSP\_rightform\_TAC simplifies a goal involving BPSP\_rightform, which asserts that a PSL process is a parameterized server process, meaning that the end of its response to each input is to call itself to wait for the next input. It does case splits on possible inputs and applies rewrites that inductively define termination and calling oneself (Terminates and Loopsback) for PSL processes. In typical cases, it automatically proves the goal.

### **B.3.9 BPSP\_invpreserved\_TAC**

BPSP\_invpreserved\_TAC simplifies a goal involving BPSP\_invpreserved, which says that if a parameterized process's state parameter satisfies a user-supplied invariant before an input event is received then the possibly changed state parameter in the process's call to itself after processing this input also satisfies this invariant. It does case splits on possible inputs and applies rewrites that inductively define the possible state parameters in the process's next call to itself (PossibleNextParameter). In typical cases, when the invariant is trivial, it automatically proves this goal.

### **B.3.10 BPSP\_nowritesdown\_TAC**

BPSP\_nowritesdown\_TAC simplifies a goal involving BPSP\_nowritesdown, which asserts that a parameterized server process does not produce any outputs in response to an input before calling itself that are at security levels not dominated by the security level of the input or are not at levels in the specified ranges. It does case splits on possible inputs and applies rewrites that inductively define "not writing down" (NoWritesDown) for PSL processes. In typical cases, it reduces this goal to showing that the levels of output events are dominated by the levels of input events.

### **B.3.11 BPSP\_nolowchange\_TAC**

BPSP\_nolowchange\_TAC simplifies a goal involving BPSP\_nolowchange, which asserts that the projection of a parameterized process's state parameter before and after it responds to an input event does not change if the projection is to a level not dominating the level of the input event. It does case splits on

possible inputs and applies rewrites that inductively define the possible state parameters in the process's next call to itself (`PossibleNextParameter`). In typical cases, it reduces this goal to showing that projections of before and after state parameters are equal.

### **B.3.12 BPSP\_samepath\_TAC**

`BPSP_samepath_TAC` simplifies a goal involving `BPSP_samepath`, which asserts that if the projections of two state parameters to a level are equal, if an input event is visible at this level, and if the process's responses to this input for these state parameters make the same nondeterministic choices, then these responses execute the same sequence of PSL commands. It does case splits on possible inputs and applies rewrites that inductively define this "same path" property for pairs of PSL processes in terms of pairs of their subprocesses (`SamePath`). In typical cases, it reduces this goal to showing equalities between boolean expressions.

### **B.3.13 BPSP\_lowresponsesame\_TAC**

`BPSP_lowresponsesame_TAC` simplifies a goal involving `BPSP_lowresponse-same`, which asserts that if the projections of two state parameters to a level are equal, then the process's responses to an input for each of those state parameters produce the same outputs visible at that level and also give state parameters for the process's next call to itself whose projections to that level are equal. It does case splits on possible inputs and applies rewrites that inductively define the possible state parameters in the process's next calls to itself (`PossibleNextParameter`). In typical cases, it reduces this goal to showing that output events and next state parameters produced for different process state parameters are the same.

### **B.3.14 BPSP\_restrictive\_TAC**

`BPSP_restrictive_TAC` is the main Romulus tactic for proving buffered, parameterized server processes restrictive. It takes the goal, rewrites it to expand out the definition of `BPSP_restrictive`, and attempts to automatically prove the three of the six subgoals that can typically be proved automatically — that the initial state parameter satisfies the invariant, the process is of

the right form, and satisfying the invariant is preserved for successive state parameters. The tactics `BPSP_nolowchange_TAC`, `BPSP_nowritesdown_TAC`, `BPSP_samepath_TAC`, and `BPSP_lowresponsesame_TAC` can be used for proving the remaining goals.

### B.3.15 ManifestlySecure\_TAC

`ManifestlySecure_TAC` is the main Romulus tactic for proving the manifest security conditions for manifestly secure processes. This tactic makes case splits on input and output events, and then rewrites with the definitions of the input and output predicates, the input and output level-assignment functions, and the dominance relation on levels. In typical cases, it reduces this goal to showing that the levels of output events dominate the levels of input events.

### B.3.16 HookupValid\_TAC

`HookupValid_TAC` is the main Romulus tactic for proving that the subcomponents of composite processes are connected together correctly. This tactic splits the goal into subgoals, and then rewrites with the definitions of the input and output predicates. For validly connected processes, the conditions that must be proved are usually trivial, and `HookupValid_TAC` typically proves the goal automatically.

## B.4 Theories

The Romulus library subdirectory `theories` contains SML source and saved-theory files for the theories made into parent theories of the new or current theory when the Romulus library is loaded. The subdirectory `src` contains the SML source, and the subdirectory `ascii` contains the saved-theory files.

The subdirectory `ascii` contains two files for each theory, a `.holsig` file and a `.thms` file. The `.holsig` file gives the constants and types defined in the theory, and the theory's parent theories. The `.thms` file gives the axioms, definitions, and theorems in the theory.

The subdirectory `src` contains SML source for each theory in the Romulus library, and a file `romutils.sml` of SML utilities that are used in defining

the Romulus libraries. These utilities include a function for automatically proving standard properties of concrete recursive types, a function for giving a more readable form of definitions for inductively defined predicates, and a function for automatically proving standard properties of inductively defined predicates.

Descriptions of each of the theories in the Romulus library follow:

#### B.4.1 romlemmas

Theory `romlemmas` contains lemmas about the standard security levels and lemmas giving simple facts of logic that are used by the Romulus tactics. The theory provides definitions of the most commonly used security levels and their dominance and less-than-or-equal-to relations, and proves useful facts about them. This is done as a convenience for the user; the user can define arbitrary sets of security levels and arbitrary order relations on them.

#### B.4.2 romproc

Theory `romproc` gives the Romulus formalization of processes. (Here “process” can refer to either a state machine or one of such a machine’s states.) It defines a particular family of atomic processes as program-like members of a concrete recursive type `PSL`, and defines transition relations specifying how these atomic processes are transformed by events. It also defines transition relations specifying how composite processes and “parent” processes serving as interfaces to other, typically composite, processes are transformed by events.

The most important part of the theory `romproc` is the following definition of the concrete recursive type `PSL` defining a particular family of atomic processes.

```
define_type {
  name = "PSL_Def",
  type_spec =
    'process = Skip |
      ;; of process#process |
      Orselect of process#process |
      If of bool#process#process |
      Send of 'outev |
      Receive of ('inev -> bool)#('inev -> 'invoc) |
```

```

      Call of 'invoc |
      Buffered of ('inev -> bool)#('inev)list#process',
fixities =
  [Prefix, Infix 1000, Prefix, Prefix, Prefix, Prefix, Prefix, Prefix];

```

Intuitive descriptions of the intended interpretations of the PSL constructors follow:

- Skip is the finished process that does nothing.
- `;;` is the “followed by” operator; `(p1 ;; p2)` is the process that does whatever `p1` does and then does whatever `p2` does.
- Orselect is the “random choice” operator; `(Orselect p1 p2)` is the process that does whatever `p1` does or whatever `p2` does.
- If is the “if-then-else” operator; `(If b p1 p2)` is the process that does whatever `p1` does if `b` is true and does whatever `p2` does if `b` is false.
- Send produces output; `(Send outev)` causes the output event `outev` to occur.
- Receive responds to input; `(Receive received response)` reacts to an input event `inev` that satisfies the predicate `received` by applying the function `response` to this input event to obtain a construct naming the process the receiving process becomes in response to this input event, then becomes this process.
- Call becomes the process named by its argument.
- Buffered is the “buffered process constructor”; `(Buffered buffering buf p)` is the same as process `p`, but buffers inputs satisfying the predicate `buffering` on a buffer whose initial contents are given by `buf`.

### B.4.3 romsecure

Theory `romsecure`, a companion theory to the process theory `romproc`, gives the Romulus formalization of security. It defines special-purpose conditions

that are sufficient to guarantee restrictiveness but easier to prove when they apply. It defines the following predicates on PSL processes:

- **Terminates.** **Terminates** tells whether a process terminates in the context of a given function assigning values to invocations. It is used only in the subsequent definition of **Loopsback**.
- **Loopsback.** **Loopsback** tells whether a process, in the context of a given function evaluating invocations, always makes a call to a particular process. The process can be non-parameterized, in which case it is given by an invocation, or parameterized, in which case it is given by a function mapping parameters to invocations. It is used to identify processes that wait for an input, process it, and return to waiting for another input.
- **NoWritesDown.** **NoWritesDown** is a relation defined for a dominance relation, a level-assignment function for outputs, a security level assumed to be the level of some input, a function mapping invocations to processes, a process name (which is either an invocation or a function mapping parameters to invocations), and a PSL process. It holds if the level of every output produced before the process ends with a call to the process given by the process name dominates the given input level.
- **PossibleNextParameter.** **PossibleNextParameter** is a relation defined for a function mapping invocations to processes, the name of a parameterized process, a parameter for this named process, and a PSL process. It holds if the parameter is possibly the parameter in the process's next call to the named process.
- **SamePath.** **SamePath** is a relation defined for a function mapping invocations to processes and two PSL processes. It holds if the executions of the two process execute the same sequence of PSL commands whenever these processes make all the same nondeterministic choices.
- **PossibleLowOutputSequence.** **PossibleLowOutputSequence** is a relation defined for a dominance relation, a level-assignment function for outputs, a security level, a list of output events, a function mapping invocations to process, the name of a parameterized process, and a PSL process. It holds if the list of output events is a possible sequence of

the output events whose levels are dominated by the given level which are produced by the process before it calls the process with the given name. Earlier output events occur more toward the list's head. It is used only in the definition of PossibleLowResponse.

- **PossibleLowResponse.** PossibleLowResponse is a relation defined for a dominance relation, a level-assignment function for outputs, a security level, a projection function, a list of output events, a parameter, a function mapping invocations to processes, the name of a parameterized process, and a PSL process. It holds if: 1) PossibleLowOutputSequence holds for the same dominance relation, level-assignment function for outputs, security level, list of output events, function mapping invocations to processes, name of a parameterized process, and PSL process; and 2) after producing this sequence of low outputs, the process calls the named process with a parameter whose projection to the security level is the parameter. One must consider the low outputs and the low next parameter at the same time, because they might not occur *together* for some processes even if both can occur separately.

In addition to defining these predicates, the theory proves, for each predicate, stronger, equational statements about the predicate that effectively define it by structural induction on the complexity of PSL objects —

```
!invocval p1 p2.
  Terminates invocval (p1 ;; p2) =
  Terminates invocval p1 /\ Terminates invocval p2
```

is such a statement. These statements are used as rewrite rules by the Romulus tactics, and doing so typically simplifies the predicates until all references to PSL have been removed. The theory also proves additional special-case statements that can be used as rewrites.

The theory defines the following non-inductive predicate and function that are convenient for expressing security properties:

- **receivesall** — a predicate that holds if and only if it is applied to a PSL Receive receiving an arbitrary input event.
- **reaction** — a partial function (actually, a conversion rule) that is applied to an input event, a function assigning PSL processes to invocations, and a PSL process. If this process is a Receive, reaction

returns the process that the `Receive` process changes to in response to the input event. If the given process is not a `Receive`, reaction is undefined and expressions involving it cannot be reduced.

Finally, the theory defines the following predicates expressing security properties. Each of these predicates assumes that the level of each input event arriving at an input port is in the level range for that port.

- **BNPSP\_rightform.** `BNPSP_rightform` is true of a predicate defining possible input events, a process and a function assigning meaning to invocations if the process receives any input event and responds to every input event by looping back to call itself. For instantiating the polymorphic predicate `Loopsback`, non-parameterized processes are treated as having the parameter (`--'one'--`).
- **BNPSP\_nowritesdown.** `BNPSP_nowritesdown` is true of predicates defining possible input and output events, and a process satisfying `BNPSP_rightform` if for every input event the process's response to that input event satisfies "no writes are down", meaning that the levels of output events dominate the level of the input event that gave rise to them and the level of every output event passing through a port is in the level range for that port. For instantiating the polymorphic predicate `NoWritesDown`, non-parameterized processes are treated as having the parameter (`--'one'--`).
- **BNPSP\_restrictive.** `BNPSP_restrictive` is a relation among predicates defining possible input and output events, a dominance relation on security levels, a function mapping input events to security levels, a function mapping output events to security levels, a function mapping invocations to processes, and an invocation. It holds if the process named by the invocation is a buffered, non-parameterized, PSL server process that is restrictive with respect to the given security-level order and level-assignment functions and whose outputs are in the required level ranges.
- **BPSP\_rightform.** `BPSP_rightform` is true of predicates defining possible input and output events, an invariant, a process, and a function assigning meaning to invocations if for every parameter satisfying the



invariant the process receives any input event and responds to every input event by looping back to call itself.

- **BPSP\_invpreserved.** BPSP\_invpreserved is true if for any parameter satisfying the invariant and any input event, any possible parameter in the process's response's call back to the process also satisfies the invariant.
- **BPSP\_nolowchange.** BPSP\_nolowchange is true if for any input event, any level not dominating the level of that event, and any initial parameter satisfying the invariant, at that level all possible parameters in the the process's next call to itself seem identical to the initial parameter.
- **BPSP\_nowritesdown.** BPSP\_nowritesdown is true if for any parameter satisfying the invariant and any input event, the response to that input satisfies "no writes are down" and the output events are in their required output ranges.
- **BPSP\_samepath.** BPSP\_samepath is true if for any level, any two parameters satisfying the invariant that seem equivalent at this level, and any input event visible at this level, if the process' responses to this input when it is parameterized by each of these parameters make the same nondeterministic choices then these responses execute the same sequence of PSL commands.
- **BPSP\_lowresponsesame.** BPSP\_lowresponsesame is true if for any level, any two parameters satisfying the invariant that seem equivalent at that level, and any input event: 1) Any possible output sequence visible at that level for one of the two parameters is also possible for the other parameter; and 2) The parameters in the process's next calls to the named process after producing these output sequences also seem equivalent at that level.
- **BPSP\_restrictive.** BPSP\_restrictive is a relation among predicates defining possible input and output events, a dominance relation on security levels, functions mapping input and output events to security levels, a projection function mapping a security level and a process parameter to the possibly sanitized version of this parameter "seen" at

that security level, an invariant on process parameters, a function mapping invocations to processes, the name of a parameterized process, and a parameter of the type appropriate for this process. It holds if the process named is a buffered, parameterized, PSL server process that is restrictive with respect to the given security-level order, level-assignment functions, projection function, and invariant and whose outputs are in their required level ranges.

#### **B.4.4    `sharedstate`**

Theory `sharedstate` is a HOL90 formalization of shared-state restrictiveness. Shared-state restrictiveness is described in Volume II, the Romulus library of models, under the generic-guard example.

#### **B.4.5    `crypto_90`**

Theory `crypto_90` is the implementation in HOL of authentication logic. This contains the primitives needed to describe and specify authentication protocols. It also contains the inference rules of the logic, which are used to carry out proofs that protocols are correct.

### **B.5    Code Executed After Loading**

After loading the Romulus utilities, tactics, and theories, loading the Romulus library executes code to bind the SML identifier `textlist` to the value `ty_antiq(==':text list'==)`. (The type `text` is defined in the Romulus theory `crypto_90`.) This variable can then be used as a type abbreviation in theories defining and analyzing protocols.

# Appendix C

## User Defined Levels

Romulus comes with a default set of levels and a default dominance relation on these levels called `standard_level` and `standard_dom`. The default levels are `top_secret`, `secret`, `confidential`, and `unclassified`, with the usual dominance relation. Romulus also allows the definition and use of alternative levels and dominance relations. This appendix shows how to do this. The example that will be demonstrated here is a set of levels that are classifications with categories with the usual dominance relations.

There are several steps that must be taken in order to define and make full use of a new set of levels.

1. Define a HOL theory describing the desired set of levels and the dominance relation on these levels.
2. Create HOL tactics for proving that one level dominates another. This step is recommended, but not required.
3. Create an `.rth` file describing the desired set of levels and the dominance relation on these levels to the Romulus graphics.
4. Create an `abbreviations` resource that contains abbreviations for the graphics to use for the new levels as described in Appendix A. This step is recommended, but not required.
5. Edit your IPSL specifications so that they use the new levels.

Each of these steps will be described in detail for the categories example.

For the categories example, we will define our new levels by adding two categories to the standard levels. A new level will then consist of a pair:

```
(standard_level, category_set)
```

where `standard_level` is a standard Romulus level (`unclassified`, `confidential`, `secret`, or `top_secret`) and the categories are from the set `{a,b}`. Some example levels are:

```
(unclassified, {})
(confidential, {a})
(secret, {a;b})
(top_secret, {})
(top_secret, {b})
```

Next we will define the HOL theory for these levels by defining a file `categories.sml` that will contain the definitions necessary for the theory. This file starts with the standard lines that remove old versions of the theory.

```
System.Unsafe.SysIO.unlink "categories.holsig"
handle e => print "no earlier categories.holsig to remove\ n";
System.Unsafe.SysIO.unlink "categories.thms"
handle e => print "no earlier categories.thms to remove\ n";
```

Next, the new theory, called `categories`, is created and the HOL set library and Romulus library are loaded.

```
new_theory "categories";
load_library_in_place(Sys_lib.set_lib);
load_library_in_place(get_library_from_disk "romulus");
```

The following creates a new enumeration type called `sample_categories` that can have value `a` or `b`.

```
val sample_categories_Def =
define_type {
  name = "sample_categories",
  type_spec = 'sample_categories = a | b',
  fixities = [Prefix, Prefix]};
```

The HOL type of a new level will be

```
(standard_level # sample_category set)
```

but we do not need to explicitly define this type. Instead we will define our new dominance relation `category_dom` using polymorphic types.

```
new_definition
  ("category_dom",
   let val category_dom =
     --'category_dom:
       (standard_level # ('category)set) ->
       (standard_level # ('category)set) ->
       bool'--;
   in
     --'
     ^category_dom (slev1,set1) (slev2,set2) =
       (standard_dom slev1 slev2) /\ set2 SUBSET set1
     '---
   end);
```

This says that one level dominates another level if the classification of the first dominates the classification of the second (using the standard Romulus dominance relation) and the set of categories of the second is a subset of the set of categories of the first.

Finally, we export the theory and exit HOL.

```
export_theory();
exit();
```

Once this file has been created the HOL theory can be created. This is easy, but takes a few minutes. The command is:

```
% rhol <categories.sml
```

If the new levels you define have a complex structure, such as the ones we have just described, it might be desirable to define HOL tactics for dealing with them. For this example, it would be convenient to define a tactic to solve subgoals of the form:

```
--'set1 SUBSET set2'--
```

A tactic that does this is easy to define:

```
val SUBSET_TAC = REWRITE_TAC [definition "set" "SUBSET_DEF"] THEN
  STRIP_TAC THEN
  REWRITE_TAC [theorem "set" "IN_INSERT"] THEN
  DISCH_TAC THEN
  REPEAT (POP_ASSUM (fn th => DISJ_CASES_TAC th) THEN ASM_REWRITE_TAC []);
```

The next tactic solves goals of the form:

```
--'category_dom level1 level2'--
```

This tactic is easily defined as:

```
val CATEGORY_DOM_TAC =  
  REWRITE_TAC [definition "categories" "category_dom"] THEN  
  CONJ_TAC THEN  
  REWRITE_TAC [definition "romlemmas" "standard_dom"] THEN  
  SUBSET_TAC THEN ASM_REWRITE_TAC [];
```

Next, it is necessary to define a file `categories.rth` that contains descriptions, in a form that can be read by the graphics, of the levels that are to be used and which levels dominate which others.

The format of this file is quite simple, even if it is quite long. It starts with the text:

```
BEGIN_ANCESTORS END_ANCESTORS
```

Next comes an enumeration of all possible levels.

```
BEGIN_LEVELS  
systemlow  
(unclassified,{})  
(unclassified,{a})  
(unclassified,{b})  
(unclassified,{a;b})  
(confidential,{})  
(confidential,{a})  
(confidential,{b})  
(confidential,{a;b})  
(secret,{})  
(secret,{a})  
(secret,{b})  
(secret,{a;b})  
(top_secret,{})  
(top_secret,{a})  
(top_secret,{b})  
(top_secret,{a;b})  
systemhigh  
END_LEVELS
```

The level `systemlow` is the lowest possible level and `systemhigh` is the highest possible level; these levels must be defined in every `.rth` file. Note that the levels defined in this file are treated by the graphics as strings, so each level must always be typed exactly as it appears in this file when you are using these levels in the graphics.

Next comes an enumeration of dominance pairs, that is, pairs of levels such that the first level dominates the second.

```
BEGIN_DOM
( systemhigh (top_secret,{a;b}) )
( systemhigh (top_secret,{a}) )
( systemhigh (top_secret,{b}) )
( systemhigh (top_secret,{}) )
( systemhigh (secret,{a;b}) )
( systemhigh (secret,{a}) )
( systemhigh (secret,{b}) )
( systemhigh (secret,{}) )
( systemhigh (confidential,{a;b}) )
( systemhigh (confidential,{a}) )
( systemhigh (confidential,{b}) )
( systemhigh (confidential,{}) )
( systemhigh (unclassified,{a;b}) )
( systemhigh (unclassified,{a}) )
( systemhigh (unclassified,{b}) )
( systemhigh (unclassified,{}) )
( systemhigh systemlow )
( (top_secret,{a;b}) (top_secret,{a}) )
( (top_secret,{a;b}) (top_secret,{b}) )
( (top_secret,{a;b}) (top_secret,{}) )
( (top_secret,{a;b}) (secret,{a;b}) )
( (top_secret,{a;b}) (secret,{a}) )
( (top_secret,{a;b}) (secret,{b}) )
( (top_secret,{a;b}) (secret,{}) )
( (top_secret,{a;b}) (confidential,{a;b}) )
( (top_secret,{a;b}) (confidential,{a}) )
( (top_secret,{a;b}) (confidential,{b}) )
( (top_secret,{a;b}) (confidential,{}) )
( (top_secret,{a;b}) (unclassified,{a;b}) )
( (top_secret,{a;b}) (unclassified,{a}) )
( (top_secret,{a;b}) (unclassified,{b}) )
( (top_secret,{a;b}) (unclassified,{}) )
( (top_secret,{a;b}) systemlow )
( (top_secret,{a}) (top_secret,{}) )
```

```

( (top_secret,{a}) (secret,{a}) )
( (top_secret,{a}) (secret,{}) )
( (top_secret,{a}) (confidential,{a}) )
( (top_secret,{a}) (confidential,{}) )
( (top_secret,{a}) (unclassified,{a}) )
( (top_secret,{a}) (unclassified,{}) )
( (top_secret,{a}) systemlow )
( (top_secret,{b}) (top_secret,{}) )
( (top_secret,{b}) (secret,{b}) )
( (top_secret,{b}) (secret,{}) )
( (top_secret,{b}) (confidential,{b}) )
( (top_secret,{b}) (confidential,{}) )
( (top_secret,{b}) (unclassified,{b}) )
( (top_secret,{b}) (unclassified,{}) )
( (top_secret,{b}) systemlow )
( (top_secret,{}) (secret,{}) )
( (top_secret,{}) (confidential,{}) )
( (top_secret,{}) (unclassified,{}) )
( (top_secret,{}) systemlow )
( (secret,{a;b}) (secret,{a}) )
( (secret,{a;b}) (secret,{b}) )
( (secret,{a;b}) (secret,{}) )
( (secret,{a;b}) (confidential,{a;b}) )
( (secret,{a;b}) (confidential,{a}) )
( (secret,{a;b}) (confidential,{b}) )
( (secret,{a;b}) (confidential,{}) )
( (secret,{a;b}) (unclassified,{a;b}) )
( (secret,{a;b}) (unclassified,{a}) )
( (secret,{a;b}) (unclassified,{b}) )
( (secret,{a;b}) (unclassified,{}) )
( (secret,{a;b}) systemlow )
( (secret,{a}) (secret,{}) )
( (secret,{a}) (confidential,{a}) )
( (secret,{a}) (confidential,{}) )
( (secret,{a}) (unclassified,{a}) )
( (secret,{a}) (unclassified,{}) )
( (secret,{a}) systemlow )
( (secret,{b}) (secret,{}) )
( (secret,{b}) (confidential,{b}) )
( (secret,{b}) (confidential,{}) )
( (secret,{b}) (unclassified,{b}) )
( (secret,{b}) (unclassified,{}) )
( (secret,{b}) systemlow )
( (secret,{}) (confidential,{}) )

```



```

( (secret,{}) (unclassified,{}) )
( (secret,{}) systemlow )
( (confidential,{a;b}) (confidential,{a}) )
( (confidential,{a;b}) (confidential,{b}) )
( (confidential,{a;b}) (confidential,{}) )
( (confidential,{a;b}) (unclassified,{a;b}) )
( (confidential,{a;b}) (unclassified,{a}) )
( (confidential,{a;b}) (unclassified,{b}) )
( (confidential,{a;b}) (unclassified,{}) )
( (confidential,{a;b}) systemlow )
( (confidential,{a}) (confidential,{}) )
( (confidential,{a}) (unclassified,{a}) )
( (confidential,{a}) (unclassified,{}) )
( (confidential,{a}) systemlow )
( (confidential,{b}) (confidential,{}) )
( (confidential,{b}) (unclassified,{b}) )
( (confidential,{b}) (unclassified,{}) )
( (confidential,{b}) systemlow )
( (confidential,{}) (unclassified,{}) )
( (confidential,{}) systemlow )
( (unclassified,{a;b}) (unclassified,{a}) )
( (unclassified,{a;b}) (unclassified,{b}) )
( (unclassified,{a;b}) (unclassified,{}) )
( (unclassified,{a;b}) systemlow )
( (unclassified,{a}) (unclassified,{}) )
( (unclassified,{a}) systemlow )
( (unclassified,{b}) (unclassified,{}) )
( (unclassified,{b}) systemlow )
( (unclassified,{}) systemlow )
END_DOM

```

Once again, the level names in this section must appear exactly the same as they did earlier. It is not necessary to list every dominance pair in this file; Romulus will compute the reflexive transitive closure of the set of pairs that are listed in this file. In other words, if A is a level then the dominance ( A A ) holds. Also, if A, B, C are levels and the dominances ( A B ) and ( B C ) hold, then the dominance ( A C ) holds. Also, the appropriate dominances for systemlow and systemhigh hold. Note that for this example many more dominance pairs are listed than are needed to define the dominance relation.

The following lines end the file.

```
BEGIN_PORTS END_PORTS
```

```
BEGIN_DISTINCT END_DISTINCT
SECURITY_PROVEN f
```

It is also convenient to define a text string that provides abbreviations for the graphics to use for each level; otherwise the full name of each level is used. For this example, the following string could be used.

```
systemlow = Lo\n\
(unclassified,{}) = U\n\
(unclassified,{a}) = U/a/\n\
(unclassified,{b}) = U/b/\n\
(unclassified,{a;b}) = U/a,b/\n\
(confidential,{}) = C\n\
(confidential,{a}) = C/a/\n\
(confidential,{b}) = C/b/\n\
(confidential,{a;b}) = C/a,b/\n\
(secret,{}) = S\n\
(secret,{a}) = S/a/\n\
(secret,{b}) = S/b/\n\
(secret,{a;b}) = S/a,b/\n\
(top_secret,{}) = TS\n\
(top_secret,{a}) = TS/a/\n\
(top_secret,{b}) = TS/b/\n\
(top_secret,{a;b}) = TS/a,b/\n\
systemhigh = Hi
```

The level names used here must appear exactly the same as they did in the .rth file.

It still remains to tell the graphics to use the levels defined in the file categories.rth and to use the abbreviations defined in the abbreviations string. This is done by setting the appropriate application resources. The Romulus installation procedure creates a directory \$ROMDIR/Environment that contains, among other things, a defaults file which will look something like this

```
*levelfile: romlemmas
*searchpath: $ROMDIR/Environment
```

where \$ROMDIR has been replaced with the full path name of the directory where Romulus was installed on your system. If you set the environment variable XENVIRONMENT with the command

```
setenv XENVIRONMENT $ROMDIR/Environment/defaults
```

then Romulus will use these application resources. To change these defaults, create your own resource file, `romenvironment`, that contains your application resources. For example, your file might look something like this:

```
*levelfile: categories
*searchpath: $ROMDIR/Environment
*abbreviations:\
systemlow = Lo\n\
(unclassified,{}) = U\n\
(unclassified,{a}) = U/a/\n\
(unclassified,{b}) = U/b/\n\
.
.
.
(top_secret,{a}) = TS/a/\n\
(top_secret,{b}) = TS/b/\n\
(top_secret,{a;b}) = TS/a,b/\n\
systemhigh = Hi
```

where `$ROMDIR` is replaced with the pathname in the original defaults file. You can also add default values for any of the application resources described in Appendix A.

You will then need to set the `XENVIRONMENT` environment variable as follows.

```
setenv XENVIRONMENT romenvironment
```

All that remains is to change your specifications to use these newly defined levels. The only special action that you need to take is to add the following lines to the `.ips1` file for the top-level component:

```
??LevelTheory: categories
??DomRelation:
category_dom:
  (standard_level # (sample_categories)set) ->
  (standard_level # (sample_categories)set) ->
  bool
??LevelVar: level:standard_level # (sample_categories)set
```

The `??LevelTheory:` entry says use the `categories` theory. The `??DomRelation:` gives the name of the dominance relation (`category_dom`) and its type. `??LevelVar:` gives the name and type of a variable denoting an arbitrary level.

Last of all, an important warning. The `ips12hol` translator cannot be used to translate `.ips1` files that have user defined levels. You will instead have to use the SPEC button. The process you need to use is this:

1. Use the Romulus graphics to create your model, save it, then exit.
2. Edit the `.ips1` files for each process to complete their specifications.
3. Start the graphics again, loading the saved model.
4. Use the SPEC button to translate the specifications.

See section 3.5.8 for more details.

# Bibliography

- [1] S. Brackin and S-K Chin. Server-process restrictiveness in HOL. In *HOL User's Group Workshop*, Vancouver, Canada, August 1993. Springer Verlag.
- [2] Stephen H. Brackin. *The Romulus Graphics Interface*. Odyssey Research Associates, Ithaca, New York, August 1992.
- [3] Stephen H. Brackin. Romulus tutorial (draft). October 1992.
- [4] Li Gong and Geoffrey Hird. The ORA toolkit for analyzing cryptographic protocols: A user manual. (Draft), 1993.
- [5] C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall International, Englewood Cliffs, NJ, 1985.
- [6] Adrian Nye and Tim O'Reilly. *X Toolkit Intrinsic Programming Manual*. O'Reilly & Associates, Inc., Sebastopol, CA, September 1990. Second Edition for X11 Release 4.
- [7] ORA. Romulus course notes, 1992.
- [8] Larry Paulson. *ML for the Working Programmer*. Cambridge University Press, 1991.
- [9] Konrad Slind. *Hol90 Users Manual, Preliminary Version*.
- [10] University of Cambridge, DSTO, and SRI International. *The HOL System Description*.
- [11] University of Cambridge, DSTO, and SRI International. *The HOL System Reference*.

- 
- [12] University of Cambridge, DSTO, and SRI International. *The HOL System Tutorial*.

# Index

- .goal.sml file, 52
  - creating, with ips12hol, 19, 25, 29
  - for simple example, 19, 22, 25, 27, 29, 32
  - for token ring example, 122
- .holsig file, 19, 22, 25, 27, 29, 32, 69
- .ips1 file, 46, 51
  - backup, 47, 51
  - for simple example, 13, 17, 18, 24, 28
  - for token ring example, 120, 122, 129, 130
- .proof.sml file
  - creating, for simple example, 32
  - creating, for token ring example, 130
  - for simple example, 22, 27
- .rom file, 46, 48
  - for simple example, 13
- .rth file, 53
  - for standard levels, 6
  - for simple example, 22, 27, 32
  - for token ring example, 131
- .sml file
  - for proving a protocol, 136
  - for specifying a protocol, 134
- .spec.sml file, *see also* HOL90, specification file, 52
  - creating, with ips12hol, 19, 25, 29
  - for simple example, 19, 25, 29, 101
  - for token ring example, 122
- .thms file, 19, 22, 25, 27, 29, 32, 69
- ;; infix operator for PSL, 99, 162
- ??Connection:, 98
- ??DomRelation:, 95, 97
- ??EndProcess:, 97, 98
- ??HOL\_functions:, 18, 95, 97
- ??InPort:, 17, 96, 98
- ??Initial:, 95
- ??Invariant:, 95
- ??LevelFun:, 18, 19, 96, 97
- ??LevelRange:, 18, 96, 97
- ??LevelTheory:, 95, 97
- ??LevelVar:, 95, 97
- ??MessageVar:, 18, 96
- ??OutPort:, 17, 96, 98
- ??Process:, 17, 94, 97
- ??ProcessInFile:, 98
- ??Projection:, 95
- ??Response:, 18, 19, 97
- ??StateVar:, 95
- [\_, \_], 10, 41
- ACCEPT\_TAC tactic, 78, 81
- analyzing flow, *see* flow analysis

- ancestors, 36
- application resource values, 141
- ASM\_REWRITE\_TAC tactic, 79, 87
- assume, 15, 50
- ASSUME\_TAC tactic, 79, 86
- asterisk
  - double, 32, 41, 131
  - single, 15, 41
- atomic processes, 1
  - security conditions, 2, 16, 23, 34, 35, 157
  - specifications, 3, 16, 35, 51, 94–97, 161
  - tactic for proving, 26, 112, 155, 160
  - translating to HOL, 19, 25, 98, 122
- authentication protocols
  - describing, 134, 135, 138
  - introduction to tools, 3
  - specifying, 134–135
  - toolkit, 133–140
  - tutorial example, 137–140
- backup function, 80, 89
- BNPSP\_restrictiveness, 108
  - \_TAC tactic, 111, 126, 157
  - predicates, 109
    - BNPSP\_nowritesdown, 109, 165
    - BNPSP\_restrictive, 165
    - BNPSP\_rightform, 109, 165
  - tactics
    - BNPSP\_nowritesdown\_TAC, 157
    - BNPSP\_rightform\_TAC, 157
  - using HOL to prove, 108–109
- BPSP\_restrictiveness, 108
  - \_TAC tactic, 111, 159
  - predicates
    - BPSP\_invpreserved, 109, 166
    - BPSP\_lowresponsesame, 110, 166
    - BPSP\_nolowchange, 110, 166
    - BPSP\_nowritesdown, 110, 166
    - BPSP\_restrictive, 166
    - BPSP\_rightform, 109, 165
    - BPSP\_samepath, 110, 166
  - tactics
    - BPSP\_invpreserved\_TAC, 158
    - BPSP\_lowresponsesame\_TAC, 159
    - BPSP\_nolowchange\_TAC, 158
    - BPSP\_nowritesdown\_TAC, 158
    - BPSP\_rightform\_TAC, 158
    - BPSP\_samepath\_TAC, 159
  - using HOL to prove, 108–111
- Buffered, 100, 162
- buffered server process
  - informal descriptions of conditions for, 107–108
- Buffered, Nonparameterized Server Process, *see* BNPSP\_restrictiveness
- Buffered, Parameterized Server Process, *see* BPSP\_restrictiveness
- Call, 99, 162
- canvas window, 6, 41
- check, 32, 53
- children, 36
- close, 46
- command
  - buttons, 6, 39
  - assume, 50



- check, 53
- close, 46
- create, 48
- create/connect, 55
- delete, 50, 57
- display, 54, 57
- flow, 45
- ipsl, 51
- load, 48
- modify, 44, 58
- move, 49, 57
- names, 47
- open, 50
- print, 47
- quit, 48
- refresh, 48
- save, 46
- spec, 51
- component level, 48
- deselecting a, 6, 39
- levels, 43
- port level, 55
- selecting a, 6, 39
- top-level, 44
- command-line options for graphical interface, 143
  - abbreviations=, 145
  - abortsavfile=, 145
  - arise=, 145
  - arun=, 145
  - bf=, 146
  - cerror=, 146
  - cf=, 147
  - cnorm=, 146
  - ef=, 147
  - initial=, 147
  - levelfile=, 148
  - lf=, 147
  - logof=, 148
  - minw=, 148
  - perror=, 149
  - pf=, 149
  - pnames=, 149
  - pnorm=, 149
  - searchpath=, 150
  - tbf=, 150
  - tef=, 150
  - tlf=, 151
  - translationstable=, 151
- component, 36
  - child, 7, 41
  - creating a, 7
  - default, 7
  - information associated with a, 36
  - naming a, 7
  - open, 41
  - proving a, 20, 25, 29
  - selecting a child, 42
  - specifying, 17, 23, 28
  - structure, 7, 41
  - top-level, 7, 41
- component commands, 43, 48
  - assume, 50
  - check, 53
  - create, 48
  - delete, 50
  - display, 54
  - ipsl, 51
  - load, 48
  - move, 49
  - open, 50
  - spec, 51, 98
  - limitations, 52

- composite processes, 1, 27–32
  - security conditions, 29, 35, 112–113, 157
  - specifications, 28, 97–98
  - tactic for proving, 31, 113, 155
  - tactics for proving, 160
  - translating to HOL, 29, 51, 53, 98
- confirming proof completion, 32, 130
- CONJ\_TAC tactic, 78, 83, 89, 90
- conventions, ii
- create, 7, 48
- create/connect, 10, 55
- crypto, 167
- cryptographics protocols, *see* authentication protocols
- delete, 50, 57
- Denning-Sacco protocol, 137
- deriving a proof for a protocol, 136
- descendants, 36
- describing a protocol, 134
  - in HOL form, 135, 138
- designing a model, 7
- DISCH\_TAC tactic, 78, 84
- display, 54, 57
- environment variables for graphical interface, 143
  - ROM\_ABBREVIATIONS, 145
  - ROM\_ABORTSAVEFILE, 145
  - ROM\_ARISE, 145
  - ROM\_ARUN, 145
  - ROM\_BF, 146
  - ROM\_CERROR, 146
  - ROM\_CF, 147
  - ROM\_CNORM, 146

- ROM\_EF, 147
- ROM\_INITIAL, 147
- ROM\_LEVELFILE, 148
- ROM\_LF, 147
- ROM\_LOGOF, 148
- ROM\_MINW, 148
- ROM\_PERROR, 149
- ROM\_PF, 149
- ROM\_PNAMES, 149
- ROM\_PNORM, 149
- ROM\_SEARCHPATH, 150
- ROM\_TBF, 150
- ROM\_TEF, 150
- ROM\_TLF, 151
- ROM\_TRANSLATIONSTABLE, 151
- EQ\_TAC tactic, 78, 84
- error messages, 42
- EXISTS\_TAC tactic, 79, 85
- exiting from the graphical interface, 13
- expand function, 80
- flow, 15, 45
- flow analysis
  - simple example, 13
  - token ring tutorial, 118
- formal specification
  - in Romulus IPSL, 17, 23, 28
  - IPSL
    - for simple example, 18, 24, 28
    - for token ring example, 120, 129
    - of atomic server process, 94
    - of composite process, 97
- frequently used tactics, 78
- g function, 80

GEN\_TAC tactic, 78, 82

goal, 77

    proved, 21

    reduced to a subgoal, 21, 26

    setting a, for a protocol, 136

    setting a, for nondisclosure, 20,  
        25, 30

goal stack, 80

graphical interface, 34

    canvas area, 6, 41

    command buttons, 6

    command levels, 43

    command-line options, 143

        -abbreviations=, 145

        -abortsavefile=, 145

        -arise=, 145

        -arun=, 145

        -bf=, 146

        -cerror=, 146

        -cf=, 147

        -cnrm=, 146

        -ef=, 147

        -initial=, 147

        -levelfile=, 148

        -lf=, 147

        -logof=, 148

        -minw=, 148

        -perror=, 149

        -pf=, 149

        -pnames=, 149

        -pnorm=, 149

        -searchpath=, 150

        -tbf=, 150

        -tef=, 150

        -tlf=, 151

        -translationstable=, 151

component commands, 43, 48

assume, 50

check, 53

create, 48

delete, 50

display, 54

ips1, 51

load, 48

move, 49

open, 50

spec, 51

editing in a text-entry window,  
    9, 40

environment variables, 143

    ROM\_ABBREVIATIONS, 145

    ROM\_ABORTSAVEFILE, 145

    ROM\_ARISE, 145

    ROM\_ARUN, 145

    ROM\_BF, 146

    ROM\_CERROR, 146

    ROM\_CF, 147

    ROM\_CNORM, 146

    ROM\_EF, 147

    ROM\_INITIAL, 147

    ROM\_LEVELFILE, 148

    ROM\_LF, 147

    ROM\_LOGOF, 148

    ROM\_MINW, 148

    ROM\_PERROR, 149

    ROM\_PF, 149

    ROM\_PNAMES, 149

    ROM\_PNORM, 149

    ROM\_SEARCHPATH, 150

    ROM\_TBF, 150

    ROM\_TEF, 150

    ROM\_TLF, 151

    ROM\_TRANSLATIONSTABLE, 151

inert areas, 42

- introduction to, 6–16
- message area, 6, 40
- parameters, 141–151
- port commands, 44, 55
  - create/connect, 55
  - delete, 57
  - display, 57
  - modify, 58
  - move, 57
- quitting, 13
- starting the, 6
- terminology, 36
- top-level commands, 43, 44
  - close, 46
  - flow, 45
  - modify, 44
  - names, 47
  - print, 47
  - quit, 48
  - refresh, 48
  - save, 46
- user defined levels, 168–177
- window, 6, 37
- X defaults files, 142

hardware requirements, i

HD, 90, 91

hd, 91

HOL90, 20, 25, 30

- ::, 63
- antiquotation operator, 69
- backward proof, 77
- cartesian products, 63
- close\_theory, 71
- cons constructor, 63
- crypto, 167
- define\_type, 73, 75
- defining HOL types and constants, 73
- describing an authentication protocol, 135, 138
- draft mode, 69
- embedding IPSL in HOL, 100
- explicitly instantiating a type, 76
- export\_theory, 71
- extend\_theory, 71
- fn, 64
- forward proof, 77
- goal file, 52
- goal stack functions, 80
- goals, 77
- hd, 62
- introduction to, 60
- invocation constructor, 105
- library, 152–167
- load\_library, 72, 153
- load\_theory, 71
- logic, 66
- mapping functions, 106
- n-tuples, 63
- new\_constant, 74
- new\_definition, 75
- new\_parent, 70
- new\_recursive\_definition, 74
- new\_theory, 70
- polymorphic type constructor, 75
- print\_theory, 72
- proof mode, 70
- record type, 63
- retrieving a definition, 72
- retrieving a theory, 72
- romcontype, 102, 155

- romgetconstant, 156
- romlemmas, 161
- romproc, 161
- romrecord, 156
- romrtheory, 156
- romsecure, *see* romsecure
- sharedstate, 167
- showing security using, 122–128
- specification file, 51, 101–107
  - created by ips12hol translator, 101–107
- tacticals, 81
- tactics, 78
- term, 66
- term\_parser, 66
- terms, 66
- theories, 69
- tl, 62
- ty\_antiq, 69
- type\_of, 67
- using, to prove BNPS\_restrictiveness, 108–109
- using, to prove BPSP\_restrictiveness, 108–111
- val, 62
- HookupValidTAC tactic, 31, 113, 155, 160
- If, 100, 162
- IMP\_RES\_TAC tactic, 79, 86
- inert areas of graphical interface, 42
- infix operator ;;, 99, 162
- input port, *see* ports
  - specification, contents of, 96
- insecure data flow, 13
- installation
  - hardware requirements, i
  - software requirements, i
- Interface Process Specification Language, *see* IPSL
- invocation, 99, 102, 106
  - constructor, 99, 105
- IPSL, 93
  - complete description, 94–98
  - introduction to, 17
- ips1, 51
- ips12hol translator, 19, 25, 29, 51, 52, 98
  - command, 122
  - HOL90 specification created by, 101–107
- justification, 78
- levels.rth file, 6
- library, HOL90, 152–167
- load, 48
- load\_library function, 153
- manifest security, 45–46
- manifest security conditions, 3, 23, 25, 129
- manifestly secure processes, 23–27
  - security conditions, 25, 35, 112, 157
  - specifying, 23–25
  - tactic for proving, 26, 112, 155, 160
- ManifestlySecureTAC tactic, 26, 112, 155, 160
- mapping functions
  - in HOL90, 106
- message window, 6, 40
- modify, 44, 58

- port operations, 10
  - top-level, 7
- move, 49, 57
- MP\_TAC tactic, 80, 89
- names, 47
- naming
  - errors, 9
  - necessity of, 7, 12
- nondisclosure security
  - simple example, 5-32
  - token ring tutorial example, 113-132
- tools
  - introduction to, 1-3
  - mechanics, 5-32
- open, 50
- ORELSE tactical, 81
- Orselect, 100, 162
- output port, *see* ports
  - specification, contents of, 96
- parent, 36
- port commands, 44, 55
  - create/connect, 55
  - delete, 57
  - display, 57
  - modify, 58
  - move, 57
- ports, 7, 10, 36, 41
  - connections between, 10, 41
  - creating, 10
  - information associated with, 37
  - naming, 12
  - selecting, 42
  - setting security-level limits, 10
  - specification, 17
- print, 47
- Process Specification Language, *see* PSL
- projection function, 107
- proving a component secure, 20, 25, 29
- proving processes secure, 107
- PSL, 98-100
  - Buffered, 100, 162
  - Call, 99, 162
  - If, 100, 162
  - infix operation ;;, 99, 162
  - Orselect, 100, 162
  - Receive, 99, 162
  - Send, 99, 162
  - Skip, 99, 162
- PSL, 35
- quit, 13, 48
- quitting the graphical interface, 13
- rated state machines, 93
- Receive, 99, 162
- refresh, 48
- REPEAT tactical, 81
- RES\_TAC tactic, 79, 87
- REWRITE\_TAC tactic, 79, 90
- rho1, 19
- romcontype, 102, 155
- romgetconstant, 124, 156
- romlemmas, 127, 161
- romlemmas.rth file, 132
- romproc, 161
- romrecord, 156
- romrtheory, 131, 156
- romsecure, 162
  - Loopsback, 163

- NoWritesDown, 163
- PossibleLowOutputSequence, 163
- PossibleLowResponse, 164
- PossibleNextParameter, 163
- reaction, 164
- receivesall, 164
- SamePath, 163
- Terminates, 163
- romulus command, 6
- Romulus-specific X application resources, 144
- rotate function, 80, 90
- rtheory file, *see also* .rth file, 53
- save, 13, 46
- save\_top\_thm function, 80
- security-level limits, 10, 41, 144
- Send, 99, 162
- set\_goal function, 80
- setting a goal, 20, 25, 30, 136
- sharedstate, 167
- Skip, 99, 162
- SML, 60
  - source code, 153
- software requirements, i
- spec, 51, 98
  - limitations, 52
  - user defined levels, 53
- specifying a component, 17, 23, 28
- specifying a protocol, 134
  - declaring initial assumptions, 135
  - postcondition, 135
- Standard ML, *see* SML
- starting the graphical interface, 6
- STRIP\_TAC tactic, 78, 85
- subcomponent, *see* component
- subgoals, 78
- tactic, 78
- tactical, 81
- tactics, 155-160
- text-entry window, 8, 39
  - editing in, 40
  - editing in a, 9
- THEN tactical, 81, 111
- TL, 90, 91
- tl, 91
- tools
  - authentication protocols, 133-140
  - authentication protocols, introduction to, 3
  - for nondisclosure security mechanics, 5-32
  - introduction to, 1-3
  - nondisclosure security, 93-111
  - introduction to, 1-3
- top-level commands, 43, 44
  - names, 47
  - close, 46
  - flow, 45
  - modify, 44
  - print, 47
  - quit, 48
  - refresh, 48
  - save, 46
- top-level component, 7, 41
- top\_goal function, 80
- translating to HOL, 19, 25, 29, 51, 52, 98
- tree address, 41
- tutorial examples
  - authentication protocol, 137-140

- nondisclosure security, 113–132
- simple nondisclosure, 5–32
- typeface conventions, ii
- user defined levels
  - translating, 53
- window
  - canvas, 6, 41
  - editing in a text-entry, 9
  - editing in text-entry, 40
  - graphical interface, 6, 37
  - message, 6, 40
  - text-entry, 8, 39
- X defaults files for graphical inter-  
face, 142
- XENVIRONMENT environment variable,  
142
- xholhelp, 92



***MISSION***  
***OF***  
***ROME LABORATORY***

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

- a. Conducts vigorous research, development and test programs in all applicable technologies;
- b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;
- c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;
- d. Promotes transfer of technology to the private sector;
- e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.